



# BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

## FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY  
A KOMUNIKAČNÍCH TECHNOLOGIÍ

## DEPARTMENT OF TELECOMMUNICATIONS

ÚSTAV TELEKOMUNIKACÍ

# TESTING THE RESPONSE OF OPERATING SYSTEMS TO DIFFERENT IPV6 FLOWS

TESTING THE RESPONSE OF OPERATING SYSTEMS TO DIFFERENT IPV6 FLOWS

## MASTER'S THESIS

DIPLOMOVÁ PRÁCE

## AUTHOR

AUTOR PRÁCE

Bc. Michal Ruiner

## SUPERVISOR

VEDOUCÍ PRÁCE

doc. Ing. Jan Jeřábek, Ph.D.

BRNO 2024

# Master's Thesis

Master's study programme **Communications and Networking (Double-Degree)**

Department of Telecommunications

**Student:** Bc. Michal Ruiner

**ID:** 220825

**Year of study:** 2

**Academic year:** 2023/24

**TITLE OF THESIS:**

## Testing the response of operating systems to different IPv6 flows

### INSTRUCTION:

Study from the literature the operation of IPv6, ICMPv6 and related protocols. Focus on the major operating systems and their most common versions and how they work with IPv6 and related protocol traffic. Study also the issues of testing the operation of these protocols and the possible differences between platforms. Also learn about existing tools for testing this issue and related vulnerabilities and attacks. As part of the thesis, develop a comprehensive list of test cases for selected tools and services for selected operating systems. These cases must then be tested on multiple versions of different operating systems. In the practical part, focus in particular on describing the testing procedure and the possible impact of any vulnerabilities. The work will also include the creation of a large-scale test environment in which the testing will be performed. The complete specification must be consulted and approved in advance by the thesis supervisor. The output of the work will also include recommendations for modifications to existing tools or the creation of tools that do not yet exist. As part of the semester project, develop the theoretical part, create a test environment and test the first tool on at least two major operating systems.

### REFERENCE:

- [1] Kurose, J. F., Ross, K. W., Computer networking: a top-down approach. 8th global ed. Essex: Pearson, 2022, 852 s. ISBN 978-1-292-15359-9.
- [2] JEŘÁBEK, J. Pokročilé komunikační techniky. Skriptum FEKT Vysoké učení technické v Brně, 2023. s. 1-179.

**Assignment deadline:** 5. 2. 2024

**Submission deadline:** 21. 5. 2024

**Head of thesis:** doc. Ing. Jan Jeřábek, Ph.D.

**Co-supervisor:** Dmitri Moltchanov

**doc. Ing. Jiří Hošek, Ph.D.**  
Chair of study program board

### WARNING:

The author of this Final Thesis claims that by creating this thesis he/she did not infringe the rights of third persons and the personal and/or property rights of third persons were not subjected to derogatory treatment. The author is fully aware of the legal consequences of an infringement of provisions as per Section 11 and following of Act No 121/2000 Coll. on copyright and rights related to copyright and on amendments to some other laws (the Copyright Act) in the wording of subsequent directives including the possible criminal consequences as resulting from provisions of Part 2, Chapter VI, Article 4 of Criminal Code 40/2009 Coll.



## **ABSTRACT**

The aim of the thesis is to create an array of virtual machines and research their response to the IPv6 protocol. Another significant part is to utilize the provided tool for generating and sniffing IPv6 traffic and verify its correct functionality. For such purpose, the GNS3 open-source software is selected. A reader is familiarized with the concepts of virtualization, GNS3 functionality and various methods of software testing together with the implemented practical models. The IPv6 protocol is introduced in detail as well as the packet format, address types and several IPv6 protocols useful for the thesis. The practical part is discussed in the Numerical results chapter. The topology is established and connectivity verified using IPv4. Configuration of static IPv6 addresses is performed on the devices as well as configuration of router to distribute particular prefixes. 5 testing scenarios are proposed that increase the input load to the tool in sense of higher number of addresses for the 3 different modes – passive, active and aggressive. 3 scripts were developed. Performance testing script measures utilization of computational resources. The other 2 scripts perform packet capturing and further analysis to compare the results of proposed scripts with provided tool. The comparison is done utilizing passive and aggressive modes. Active mode is used to observe the response of various operating systems to different IPv6 flows. Specifically, multiple Windows 10 builds, Linux distributions, Windows XP, 7, 11, macOS and Android.

## **KEYWORDS**

Virtualization, GNS3, Software testing, IPv6, ICMPv6, Neighbor Discovery, SLAAC, MLD, mDNS, LLMNR, Operating system, Network discovery, Network scanning, Performance evaluation

RUINER, Michal. *Testing the response of operating systems to different IPv6 flows*. Master's Thesis. Brno: Brno University of Technology, Faculty of Electrical Engineering and Communication, Department of Telecommunications, 2024. Advised by doc. Ing. Jan Jeřábek, Ph.D.

# Author's Declaration

**Author:** Bc. Michal Ruiner  
**Author's ID:** 220825  
**Paper type:** Master's Thesis  
**Academic year:** 2023/34  
**Topic:** Testing the response of operating systems to different IPv6 flows

I declare that I have written this paper independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the paper and listed in the comprehensive bibliography at the end of the paper.

As the author, I furthermore declare that, with respect to the creation of this paper, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll. of the Czech Republic, Section 2, Head VI, Part 4.

Brno .....

.....

author's signature\*

---

\*The author signs only in the printed version.

## ACKNOWLEDGEMENT

I would like to thank the advisors of my thesis, doc. Ing. Jan Jeřábek, Ph.D., Dr. Dmitri Moltchanov and Prof. Evgeni Kucheryavy for their valuable comments etc.

# Contents

<b>1</b>	<b>Introduction</b>	<b>13</b>
1.1	Research questions . . . . .	14
<b>2</b>	<b>Software and methodology</b>	<b>15</b>
2.1	Virtualization . . . . .	15
2.1.1	Virtualization implementation . . . . .	15
2.1.2	Types of virtualization . . . . .	16
2.2	GNS3 . . . . .	17
2.2.1	Components . . . . .	18
2.2.2	Real-time analysis . . . . .	18
2.2.3	Get virtual machine to the topology . . . . .	18
2.2.4	Remote server solution . . . . .	19
2.3	Software testing . . . . .	20
2.3.1	Software Testing Methodologies . . . . .	21
2.3.2	Models in development . . . . .	22
<b>3</b>	<b>Internet Protocol version 6</b>	<b>24</b>
3.1	IPv6 packet . . . . .	25
3.2	IPv4 and IPv6 datagram differences . . . . .	26
3.3	IPv6 addresses . . . . .	27
3.3.1	Types of addresses . . . . .	28
3.3.2	Subtypes of unicast addresses . . . . .	28
3.3.3	Multicast addresses . . . . .	30
3.4	Internet Control Message Protocol version 6 . . . . .	31
3.5	Neighbor Discovery for IPv6 . . . . .	32
3.5.1	Neighbor MAC address resolution . . . . .	32
3.6	Automatic address configuration . . . . .	34
3.6.1	SLAAC . . . . .	34
3.6.2	DHCPv6 . . . . .	36
3.7	Multicast Listener Discovery . . . . .	37
3.7.1	MLDv1 . . . . .	38
3.7.2	MLDv2 . . . . .	39
3.8	IPv6 in operating systems . . . . .	40
3.8.1	Linux . . . . .	41
3.8.2	Windows . . . . .	41
3.8.3	macOS . . . . .	42

<b>4</b>	<b>Numerical results</b>	<b>43</b>
4.1	ptnetinspector tool . . . . .	43
4.2	Testing topology . . . . .	44
4.2.1	Connectivity check . . . . .	44
4.2.2	IPv4 dynamic configuration . . . . .	50
4.2.3	IPv6 SLAAC configuration . . . . .	51
4.3	Scripts . . . . .	52
4.3.1	Performance testing script . . . . .	52
4.3.2	Traffic capturing script . . . . .	54
4.3.3	Verification of results script . . . . .	55
4.4	Application testing methodology . . . . .	60
4.4.1	Testing of Passive mode of the ptnetinspector . . . . .	60
4.4.2	Testing of Active mode of the ptnetinspector . . . . .	64
4.4.3	Testing of Aggressive mode of the ptnetinspector . . . . .	72
4.5	ptnetinspector adjustments . . . . .	80
4.5.1	mDNS payload reading . . . . .	81
4.5.2	Timestamp addition . . . . .	83
4.5.3	Google public DNS address . . . . .	84
4.6	Large-scale testing . . . . .	85
4.6.1	Passive mode testing inside BUT network . . . . .	86
4.6.2	Active mode testing of various Windows 10 builds . . . . .	87
4.6.3	Active mode testing of various operating systems . . . . .	90
4.6.4	Aggressive mode testing of various operating systems . . . . .	91
4.6.5	Scanning vulnerability . . . . .	92
	<b>Conclusion</b>	<b>95</b>
	<b>Bibliography</b>	<b>98</b>
	<b>Symbols and abbreviations</b>	<b>102</b>
	<b>List of appendices</b>	<b>105</b>
	<b>A Verification of results script code sample</b>	<b>106</b>
	<b>B Performance testing script code sample</b>	<b>108</b>
	<b>C Traffic capturing script code sample</b>	<b>109</b>
	<b>D Content of the electronic attachment</b>	<b>110</b>

# List of Figures

2.1	GNS3 environment . . . . .	17
2.2	Device communication between local and remote server . . . . .	20
3.1	Basic header structure of IPv6 packet . . . . .	25
3.2	Basic structure of ICMPv6 message . . . . .	31
3.3	Basic structure of ICMPv6 Neighbor Solicitation message . . . . .	32
3.4	Basic structure of ICMPv6 Neighbor Advertisement message . . . . .	33
3.5	Basic structure of ICMPv6 Router Advertisement message . . . . .	35
3.6	Basic structure of MLDv1 packet . . . . .	38
3.7	Basic structure of MLDv2 Report packet . . . . .	39
4.1	Designed topology in GNS3 for the proposed testing scenarios implementation . . . . .	44
4.2	Designed topology in GNS3 for the operating systems IPv6 response research . . . . .	45
4.3	Ping command issued from Kali machine . . . . .	49
4.4	Captured packets of Kali ping in Wireshark . . . . .	50
4.5	R1 DHCP binding . . . . .	52
4.6	Active mode scenario 5 – performance test script output . . . . .	53
4.7	Designed performance testing script flowchart . . . . .	54
4.8	Designed traffic capturing script flowchart . . . . .	56
4.9	Verification of results script code structure . . . . .	57
4.10	Designed verification of results script flowchart . . . . .	59
4.11	Program output of the passive mode scenario 2 – hosts with Link-Local addresses only . . . . .	62
4.12	Program output of the passive mode scenario 3 . . . . .	63
4.13	Passive mode scenario 3 – conversation sample in Wireshark . . . . .	63
4.14	Program output of the passive mode scenario 4 . . . . .	65
4.15	Program output of the passive mode scenario 5 for Ubuntu . . . . .	66
4.16	Program output of the passive mode scenario 5 for router . . . . .	66
4.17	Kali automatically generated IPv6 addresses . . . . .	67
4.18	Program output of the active mode scenario 1 . . . . .	67
4.19	IPv6 conversations during active scan in the scenario 1 . . . . .	68
4.20	IPv6 conversations during active scan in the scenario 2 . . . . .	68
4.21	Program output of the active mode scenario 3 . . . . .	69
4.22	IPv6 conversations during active scan in the scenario 3 . . . . .	70
4.23	Program output of the active mode scenario 4 . . . . .	71
4.24	IPv6 conversations during the active scan in the scenario 4 . . . . .	72
4.25	Program output of the active mode scenario 5 (Windows 11) . . . . .	73

4.26 IPv6 conversations during the aggressive scan in the scenario 1 . . . . .	74
4.27 Program output of the aggressive mode scenario 2 . . . . .	74
4.28 IPv6 conversations during the aggressive scan in the scenario 2 . . . . .	75
4.29 Program output of the aggressive mode scenario 3 . . . . .	76
4.30 IPv6 conversations during aggressive scan in the scenario 3 . . . . .	77
4.31 Program output of the aggressive mode scenario 4 . . . . .	79
4.32 IPv6 conversations during the aggressive scan in the scenario 4 . . . . .	80
4.33 Program output of the aggressive mode scenario 5 (clients) . . . . .	81
4.34 Program output of the aggressive mode scenario 5 (router) . . . . .	82
4.35 mDNS APDU not processed in the passive mode by the ptnetinspector	82
4.36 mDNS addresses not processed in the passive mode by the ptnetinspector – Wireshark output . . . . .	83
4.37 End time of the ptnetinspector process . . . . .	84
4.38 Start and end time of the ptnetinspector packet capturing . . . . .	84
4.39 ptnetinspector Google DNS address assigned to the router . . . . .	85
4.40 Google address ignored in the verification script . . . . .	85
4.41 Passive mode testing results with 1 device missed by the ptnetinspector	87
4.42 Start and end timestamps stated by the ptnetinspector . . . . .	88
4.43 Packets captured by the designed script with omitted source MAC address by the ptnetinspector . . . . .	88
4.44 Actual packets captured by the ptnetinspector . . . . .	88
4.45 LLMNR and mDNS messages generated by Windows 11 during booting	91
4.46 Aggressive mode testing – ignored Google DNS address by the designed script . . . . .	92



# List of Tables

4.1	GNS3 VM addresses – testing scenarios implementation . . . . .	45
4.2	GNS3 VM addresses --various operating systems . . . . .	46
4.3	GNS3 VM addresses --Windows 10 builds . . . . .	47
4.4	Performance testing of the active mode – 3 runs measured . . . . .	72
4.5	Performance testing of the aggressive mode – 3 runs measured . . . . .	80
4.6	Passive mode testing inside BUT network . . . . .	86
4.7	Windows 10 Pro versions included in the active mode testing . . . . .	89
4.9	Various OS versions included in the active mode testing . . . . .	90
4.11	Aggressive mode testing with virtual machines . . . . .	92
4.8	Windows 10 various build responses in the active mode . . . . .	93
4.10	Various OS responses in the active mode . . . . .	94

# Listings

A.1	Verification of results Python script code sample . . . . .	106
B.1	Performance testing Bash script code sample . . . . .	108
C.1	Traffic capturing Bash script code sample . . . . .	109

# 1 Introduction

On the internet, each device is uniquely identified with a logical address corresponding to the network layer of both ISO/OSI and TCP/IP models. The IPv4 protocol was used for a long time, but with advancements in the technology, the development led to more and more devices connected to the internet, which made the IPv4 address space insufficient. Although some mechanisms were introduced to make the depletion of addresses slower, it was still inevitable and a new protocol had to be designed as a response. The IPv6 was introduced in 1998. Apart from the much larger address space ( $2^{128}$  against the  $2^{32}$  of IPv4), the new ways of thinking have emerged, also influenced by the experience gained over time, which led to the changes in header fields. Although the original intention was to fully replace IPv4 protocol, it is still used widely today and the complete replacement is not probable in the near future. There are certain mechanisms how to make these 2 protocols coexist, however more and more companies providing services move to IPv6 (either only or next to IPv4), which in turn creates demands on the developers of operating systems for end hosts to implement the support of IPv6 protocol. The aim of this thesis is to create the testing array of various operating systems (different versions and distributions) and research their response to the IPv6. The tool *ptnetinspector* is used for this purpose and must be verified.

The second chapter focuses on the software and methodologies used in the thesis. Virtualization together with GNS3, the software used to create the virtual machine testing array, are introduced. The components, real-time analysis, adding virtual machines to the topology and remote server solutions are described. Then the software testing is discussed. Basic testing types, methods for running tests and development methodologies and models are covered. The third chapter focuses on the IPv6 protocol. IPv6, packet structure, various types of addresses and some of the IPv6 protocols, namely ICMPv6, ND and MLD are described. The chapter also covers automatic configuration and IPv6 implementation in common operating systems. The fourth chapter focuses on practical results. Testing topology is introduced together with the process of testing connectivity using IPv4 protocol and the configuration of SLAAC. Then the developed scripts for the *ptnetinspector* tool are described. These are Bash script designed for the resource utilization calculations, Bash script developed to capture the network traffic and the Python script used to analyse and compare the results with *ptnetinspector*. 5 scenarios were proposed to verify the tool running in 3 modes – passive, active and aggressive. Finally, the scripts are compared under heavy load inside real BUT Wi-Fi network (passive mode) and virtual environment (aggressive mode). Active mode was utilized to research the response of various operating systems to different IPv6 flows.

## 1.1 Research questions

In the thesis, following research questions are solved and the answers to them are summarized in the Conclusion chapter:

RQ1: *Are there any detection errors or other bugs in ptinetinspector tool provided for IPv6 devices discovery? Is there a way to resolve them?*

RQ2: *What is the performance of ptinetinspector tool under certain scenarios?*

RQ3: *What, if any, are the differences between the selected major operating systems in terms of specific aspects of IPv6 implementation?*

The RQ1 is specifically addressed in the section 4.5, RQ2 in the section 4.3.1 and sections 4.6.2 and 4.6.3 provide answers to the RQ3.

## 2 Software and methodology

In this chapter, the software utilized in the thesis as well as premises and methodologies are described.

### 2.1 Virtualization

Virtualization is a process of creating single or multiple virtual instances of computers, also called VM (Virtual machine), with operating systems on top of one physical device. It can be also seen as the emulation of an operating system. This leads to one extreme benefit, that is cost savings. One powerful physical device can be purchased and its computational resources - CPU (Central Processing Unit) cores, RAM (Random Access Memory), storage space either HDD (Hard Disk Drive) or SSD (Solid State Drive) - can be redistributed across many VMs. Other benefits might be efficient way of using resources (one physical device could not use all the possible performance, on the other hand multiple VMs could use up to 100 % of it) and reduced downtime (in case of OS (Operating System) corruption, other VM can be run as a copy of the original instance and the end user just shifts its work). From the point of view of VM, there is no difference between running the actual OS in a standard manner or inside a virtual environment [1], [2].

#### 2.1.1 Virtualization implementation

To be able to apply virtualization, one needs a **hypervisor**. Hypervisor is a piece of software that creates additional layer in the standard layer model (Hardware → BIOS → OS). It takes the responsibility for resource allocation among the VMs and their correct functioning. Two types of hypervisors exist:

- **Type 1 hypervisor** – Also called bare-metal, runs directly above a hardware (Hardware → BIOS → Hypervisor → VM). In comparison with the Type 2, more efficient way of virtualization (resource usage) is possible as there is direct access to the physical resources. To create and manage the VMs, the **management console** is needed. It allows an administrator to have control over the entire environment and also perform the **resource over-allocation**, which means that more resources than the actual (physically) available are assigned under assumption that there won't be a state where the VMs would use all the resources at the same time. However, in practise only the amount of resources actually needed by the VMs to ensure correct functioning is assigned by the hypervisor. A VM can be also easily moved to another physical device. (as the hardware is abstracted, management console takes care of it). This type is

typical in data centers and companies offering cloud services. The examples of Type 1 hypervisors are VMware ESXi, KVM (Kernel-based Virtual Machine) and Microsoft Hyper-V [1], [2].

- **Type 2 hypervisor** – Also called hosted hypervisor, is an application installed on the hosting OS (Hardware → BIOS → Host. OS → Hypervisor → VM). This type is more common at the end user level, as it is easy to use and often only several virtual machines are needed for the purpose of testing or security reasons. One disadvantage when compared to the Type 1 is that additional overhead in terms of performance is added as the virtualized system must access the hardware resources using the host OS as the relay system. The available hypervisors on the market are VMware Workstation/VMware Workstation Player and Oracle VM VirtualBox [1], [2].

To decide which hypervisor is more appropriate, one needs to take into account available physical resources and financial budget, the purpose of virtualization, size of the topology and many other aspects [1], [2], [3].

## 2.1.2 Types of virtualization

So far, the 2 types of virtualization were discussed, that is server virtualization (using Type 1 hypervisors) and desktop virtualization (using Type 2 hypervisors), but there are also other types:

- **Storage virtualization** – All the available storage space across the network is virtualized and accessed as one single storage unit.
- **Application virtualization** – Only an application, not the whole OS, is running in the virtual environment (different approaches exist; can be run locally or on a remote server).
- **Network virtualization** – Physical devices (e.g. routers, switches, firewalls) that may be part of separate networks are bundled into one virtual network or the other way around. An example can be configuration of VLAN (Virtual Local Area Network) or VPN (Virtual Private Network). Security of networks is improved this way [4].

Additionally, the network components can be abstracted and integrated to the software running on a hypervisor, creating a central location to access devices and modify their configuration. This brings ease of administration for the network engineers.

These are just some of the available virtualization types used. There are many others used in different branches of IT (Information Technology), among them CPU and GPU (Graphics Processing Unit) virtualization and Cloud virtualization.

## 2.2 GNS3

GNS3 (Graphical Network Simulator-3) is an open source software, more specifically emulator, used by the network engineers all around the world to emulate networks, test the functionality of designed topologies, check the proposed configurations etc. The GUI (Graphical User Interface) of the software can be seen in the Fig. 2.1.

As comes from its properties, the source code of the software is publicly available on GitHub, from where it can be downloaded and compiled or there is an option to download the executable file for installation. All the users have the possibility to propose their own improvements and contribute to the community by editing the code and implementing their ideas. The emulator means that the hardware can be virtualized and the real software images run on the devices in a topology.

The development of GNS3 started more than 10 years ago by Jeremy Grossman to help him with his Cisco exams, therefore mainly Cisco devices were intended to be in use. Since then, by publicly releasing the project and thanks to the cooperation of still growing community, many vendor devices are now supported, including MikroTik, Juniper, FortiGate and many operating systems including Windows (XP, 7, 8, 10, but also servers), Linux distributions (e.g. Kali, Ubuntu, CentOS) and other appliances available from the official marketplace [5].

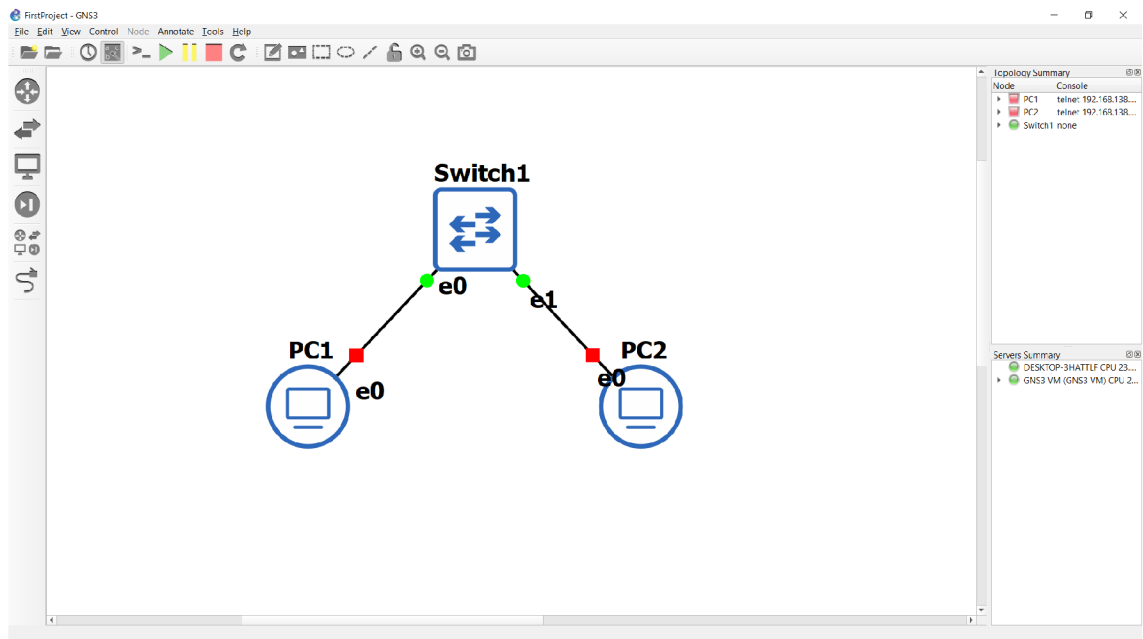


Fig. 2.1: GNS3 environment.

Regarding the Cisco IOS images, GNS3 can emulate the hardware, but the software cannot be provided legally due to the copyright. Here comes one of the most important advantages of GNS3: it works with real images and therefore provides real picture of functionality. Compared to the "simple" network simulator, just as Cisco Packet Tracer, where there is only certain number of commands available and still only simulated behavior is provided based on the predefined calculations, emulator provides all the possibilities provided on the real devices. The other advantage is that the topology created in GNS3 can be connected to the real network and then the traffic flows through the virtualized network to the internet and back. One can also make the real traffic generated on the physical computer to flow through this network and intentionally affect the delay or throughput.

### **2.2.1 Components**

GNS3 offers 2 main components: application itself and VM. After the installation, only the local server instance, through which all the appliances just as VPC (Virtual Personal Computer), switches, routers and other stuff run, is available. But there are other options like running locally or remotely GNS3 VM. It is recommended specifically for Windows users to install VM as well, because there are certain benefits like possibility to run more devices and have larger topologies. This is not the case of Linux users. The VM instance must be run through virtualization software (called hypervisor) such as VMware or VirtualBox. For a certain time, VMware was preferred because VirtualBox did not support nested virtualization (virtual machine can run other virtual machine via its own hypervisor), but now, this is no longer a concern [5].

### **2.2.2 Real-time analysis**

Another great feature of GNS3 is that it allows capturing packets of the network interfaces by the network analyzers such as Wireshark. This further improves engineer's ability to perform troubleshooting. Users can select individual interfaces to monitor and go through individual messages sent, check the formatting, addresses and observe if devices of different vendors can communicate with each other. Such possibility together with enabled logging on the devices makes great way of learning, understanding and analyzing the network behavior.

### **2.2.3 Get virtual machine to the topology**

There are more ways of importing a VM to the GNS3 environment. One of them is to visit the GNS3 marketplace, section Appliances, and download a prepared



appliance, then get one of the supported image (OS) versions and install it to the GNS3. Cisco routers can be also imported as Dynamips appliances with particular image<sup>1</sup>.

The other option, which is also used for this thesis, is to preinstall a VM in any hypervisor and import it to the GNS3 project. VMware Workstation Player was chosen for the reason described earlier. First, virtual machine is created and resources of the computer are allocated. Then the OS (commonly ISO file) is connected via CD/DVD and installed. This way all the VMs were installed, but there was slight complication that Windows 11 needs TPM (Trusted Platform Module) module for security-related features, but only encryption of certain parts can be set.

To import VM to the topology, one must choose correct hypervisor (QEMU, VirtualBox, VMware) and add it as a new machine. This way each VM was successfully added, but there was one exception. Current version of GNS3, 2.2.43, cannot run Windows 11 with the TPM module set. If one tries it, he will experience timeout. Simple solution for this is, after the installation in VMware, to remove TPM module and cancel encryption. Windows 11 will still work and GNS3 will be able to run it as regular VM. According to the regular updates, GNS3 added support for TPM module to the QEMU VMs [6].

After the Windows XP Professional installation, the Intel update (Ethernet Network Adapter Driver 18.0) was necessary so that the adapter is available on the machine. Regarding IPv6, only the basic TCP/IP stack with IPv4 protocol was accessible, therefore to enable IPv6 support, the following command must have been issued to install the TCP/IP IPv6 stack:

```
1 netsh interface ipv6 install
```

For the macOS Monterey, it is necessary to download and install patch tool[7] that unlocks an option to install various versions of macOS in VMware. According to the author, the latest version of VMware Workstation Player that the unlocker was tested on was version 15, but it also works correctly on the version 17.5.0 (version of VMware used in this diploma thesis).

The Android VM utilized in the diploma thesis is Android x86[8] (an open source project used to make Android Open Source Project runnable on the x86 platform) version 9.0.

## 2.2.4 Remote server solution

GNS3 allows users to add remote server to the project along with local and GNS3 VM servers. This solution is efficient when local machine resources are limited and

---

<sup>1</sup>This way Cisco router was added to the testing topology in this thesis.

there is necessity to have large topology that overreaches its capabilities. In this thesis, remote server from the BUT network was added to the topology.

To enable communication between machines running on the local server and those running on the remote server, local server binding must be edited in the settings of GNS3 (set the particular interface). In case of the added BUT server, the interface going to the VPN tunnel must be chosen (same network).

The demonstration example is shown in the Fig. 2.2. Kali Linux, running on the local server, and Arch Linux, running on the remote server, were added to the topology and their connectivity (ping using the Link-Local address) was tested as can be seen in the figure.

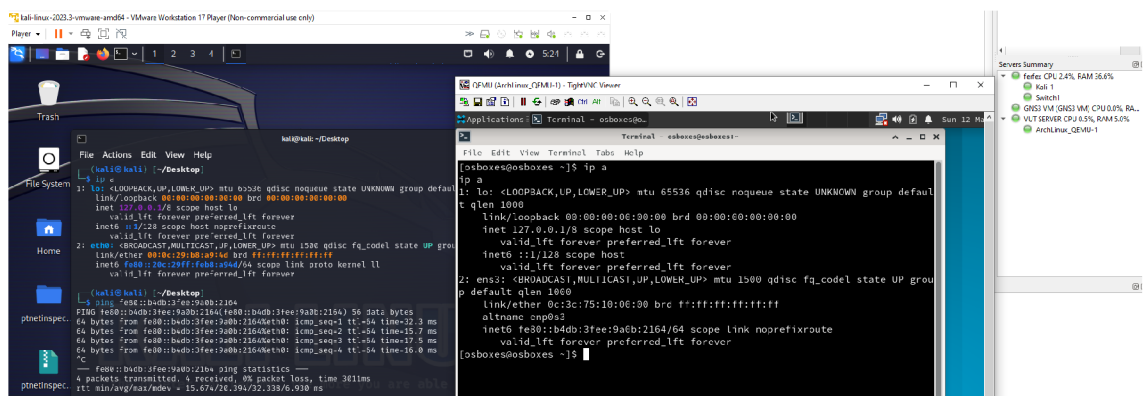


Fig. 2.2: Device communication between local and remote server.

The Kali machine runs in VMware. Arch Linux is added to the remote server as QEMU virtual machine. This VM was taken from the [www.osboxes.org](http://www.osboxes.org) page as a prepared VMware virtual disk file. Other Linux machines can be also found there. During the attempts to run Windows VMs (created locally in VMware) in the same way as Arch Linux, it was discovered that mere *vmdk* file is not sufficient as it is not recognized as bootable device. The situation did not change even after conversion to the QEMU image file format *qcow2*. Possible solution is to create VM directly in QEMU and upload the resulting virtual disk to the remote server.

## 2.3 Software testing

Software testing is a necessary part in software design. The behavior of a program must be checked against the expected results. It is not possible to publish the project right after finishing all the parts, because users might experience faulty behavior. This would influence their work efficiency and they might choose competitor's product instead. Program could contain more or less severe bugs, wrong calculations,

endless looping etc., which might diminish the company's profits. To address this, initial tests must be performed to ensure correctness, high performance, security and many other aspects related to the particular software.

There are tens of types of testing, but not all are applicable generally. First of all, **Alpha testing** and **Beta testing** can be distinguished. Alpha testing is the first phase performed internally by the company. Specialized teams (or hired companies) can be allocated for such purpose. The goal is to catch potential misbehavior and report it to the developer teams so that errors can be fixed. But not only the bugs are searched for, the time aspect is also important. The overall runtime of a program and the calculation time of specific operations could have severe consequences if they are excessively long. After passing all the tests and obtaining acceptable results, Beta testing follows. Product is released to the end users, which in turn report the bugs or features they are not satisfied with. This approach might be only high-level as the software can be proprietary and the users would not have access to the source code. Another case is open-source code, where users can try to fix the bugs themselves or add features [9].

Tests can be run either **manually** or be **automated**. In case of manual testing, the programmer performs tests by himself without the usage of any automated tool. This is an effective way to test individual parts, such as functions, on the go, but it requires understanding the code and enough knowledge how to perform such tests. For the automated testing, a special tool is used that performs the tests automatically without much intervention from the programmer side. Sets of data for testing might be prepared in advance and finally compared to the expected results. This approach can save human, time and financial resources once the tool (particular script or sequence of scripts) is prepared. [10], [11].

### 2.3.1 Software Testing Methodologies

The methodologies for software testing define various approaches to verify the correct behavior of a program. At the output, it should be determined whether the results are as expected, no errors are recorded during execution, the program offers high enough performance and it fulfills requirements of the original assignment.

From the high level point of view, the software testing is divided into 2 categories:

- **Functional testing** – This category focuses on testing the functionalities, that were specified in the beginning of the project. This could be the client's software requirements or developer's intentions. Basically, 4 types of testing fall into this group:
  - **Unit testing** – Developers create these tests to verify the functionality of individual blocks the whole project is composed of. Individual functions

have their own unit tests which verify correct results.

- **Integration testing** – After functions successfully pass the unit tests, they are then integrated together and checked to ensure they work properly, i.e. verifying the interactions between those functions.
- **System testing** – When all the individual components (or blocks) are combined to form the final application, they are tested as a whole system by providing the input and comparing the actual output to the expected output.
- **Acceptance testing** – The application as a whole is compared to the initial specifications and checked if all the desired features are implemented. These tests are typically performed by the intended users [12], [13].
- **Non-functional testing** – This group of tests checks other than functional aspects of the software, such as operational, which are the following:
  - **Performance testing** – The behavior of application is evaluated against varying inputs, such as increasing volumes of input data. Similar to this, processing time of the software can be measured when exposed to regular or excessive load.
  - **Security testing** – Vulnerabilities that could be exploited by the malicious users are detected and mitigated or better completely removed. Application must be resistant against the sensitive data breaches and the unauthorized access. Properties such as a data integrity and confidentiality can be achieved by the usage of specific cryptographic algorithms. Penetration testing (simulation of real-world cyberattack against the application with the purpose of finding vulnerabilities) falls to this category.
  - **Usability testing** – This kind of testing evaluates the user experience from using the application. The assessed aspects are, for example, ease of use or effectiveness (how well the tasks can be accomplished by the users).
  - **Compatibility testing** – Compatibility is crucial for expanding the user base. This kind of testing aims to verify the correct functioning of the application across the devices from multiple vendors and across the different operating systems [12], [13].

### 2.3.2 Models in development

There exist several models used for the software development in practise:

- **Waterfall model** – This model consists of 5 stages which go one after the other. The succeeding one must wait till the preceding one is completed. The phases are in the following order: Requirements, Design, Implementation,

Verification, Maintenance. During the first phase, detailed plan for testing is created based on the original requirements and objectives. Next, the testing tools and various scenarios to check the correct functionality are designed based on the Requirements phase and then implemented. After that, functionality of the software as a whole is verified by the customer and then the company provides support and constantly releases updates for the software. This model is suitable for small projects where applications are not very complex.

- **Iterative model** – The development of a project is split into smaller parts, which are iteratively developed (similar phases to Waterfall model) and tested before finally merging to the main application. After merging all the individual blocks, the application is tested as a whole. Results of each cycle are important for the subsequent cycle as the analysis can provide view on what can be further improved. This allows for incremental development and flexible design.
- **Agile model** – Unlike Waterfall model, Agile model is useful for larger projects with higher level of complexity and can address projects where the requirements change in time. The software development is divided into iterations and the final product is built up gradually. Each iteration adds certain improvements or features that enhance the overall functionality of the software. Such features are tested before merging them into the main application. Use case of such model might be when the project requirements are not known in advance and customer shapes his idea on the go. In such cases, the application can work with its core functionality while additional features are implemented and improved on an ongoing basis.
- **Extreme Programming model** – This model is very similar to the Agile model as it adopts the same principles. Developers present simple code, if it passes the tests and is approved, they proceed to further tasks. Basically, 2 entities work on the same feature where one is doing the programming part and the other performs checks, testing and comes with new ideas.
- **Verification and Validation model** – This model basically extends the Waterfall model. In the first phase, all the requirements are verified, including the design patterns. After finishing the planning, the second phase, implementation, begins. Results of the first phase are implemented and finally the third phase validates the results according to the expected behavior and determined requirements in the verification phase [14], [15], [16].

## 3 Internet Protocol version 6

IPv6 (Internet Protocol Version 6) is a new protocol defined on the network layer of the TCP/IP network model. It does not just represent single protocol, but it is more a set of the completely new defined protocols, which are more or less developed from their previous IPv4 versions and modified to better reflect demands of today's networks<sup>1</sup>. IPv6, as the network layer protocol itself, is defined in the RFC 2460 [17].

The need for introducing a new protocol raised initially from the insufficient address space of the former protocol. As the IPv4 addresses are 32 bits long, the IPv4 address range is  $2^{32}$  ( $\approx$  4,2 billion of addresses), while the IPv6 address range is  $2^{128}$  ( $\approx$  340 undecillion of addresses). One can easily determine that with growth of the mobile networks, IoT (Internet of Things) and other technological advancements that had caused rapid use of addresses, the IPv4 address space is insufficient and it was already depleted in 2011. There were some attempts to slow down the depletion, like for example NAT (Network Address Translation), but it was only a matter of time before this approach was further ineffective.

Even though IPv6 solves the lack of addresses, the significant part of the world is still functioning on IPv4 and this cannot be omitted. It is not possible (and probably won't be for several decades) to simply remove IPv4 and therefore mechanisms to ensure coexistence of these 2 protocols are necessary (as IPv4 and IPv6 are not compatible).

First effective method to deploy is **dual stack**, where devices have configured both IPv4 and IPv6 addresses. This can bring many advantages, among them each of the protocols runs separately and if there will be time when IPv4 will be fully replaced, it can be just removed from the configuration. On the contrary, demands on devices in terms of memory bring disadvantage as everything must be doubled, including routing tables.

The other way to ensure coexistence is **tunneling**. This way only one protocol is configured in the network and the latter is tunneled. For example, if IPv4 protocol is configured and there are IPv6 packets to be transmitted, new packets with IPv4 header are assembled and the original IPv6 packet becomes its payload. It is possible to set up tunnels automatically or manually, but both methods require certain knowledge, which can be either time consuming for the administrator or consume resources of intermediate devices for the tunnel establishment.

Another approach is **protocol translation**, that works in the similar way to

---

<sup>1</sup>Since IPv4 was first defined in 1980, the assumptions for address space and network functioning were quite different from today's reality. Everything is developing and so the technology must reflect most current needs.

standard NAT. One protocol address (IPv6) is translated to the other one (IPv4) and the other way around. This enables communication between IPv6-only and IPv4-only devices. NAT64 is the protocol used for such purpose. It is closely associated with DNS64, which provides artificially created IPv6 address of the destination possessing only IPv4 [18].

### 3.1 IPv6 packet

IPv6 format follows IP standard on the network layer, meaning it consists of a header and a payload. Header conveys information for the intermediate nodes to decide about routing, priority operations and other stuff. Useful data are transmitted inside payload. The structure is displayed in the Fig. 3.1. The header has fixed length of 40 B.



Fig. 3.1: Basic header structure of IPv6 packet.

The header consists of the following items:

- **Version** – 4-bit item which specifies version of the IP protocol. Decimal value 4 (*0100b*) is set for IPv4 and value 6 (*0110b*) for IPv6.
- **Traffic Class** – 8-bit value where the originator specifies class (or priority of the packets) for the intermediate devices to prioritize some packets over the others. This way QoS (Quality of Service) can be achieved. First 6 bits are used for the DSCP (Differentiated Services Code Point) and 2 bits for the ECN (Explicit Congestion Notification).
- **Flow Label** – 20-bit value where the originator specifies which packets belong to the same flow and the same processing is desired. This way packets can be routed the same way or a special treatment based on the Traffic Class field is acquired. Value 0 means that a packet does not belong to a specific flow.
- **Payload Length** – 16-bit value allows to carry payload (data from the upper layers) of up to 64 kB. In case extension headers are utilized in the packets, they are part of payload. The length is in bytes.
- **Next Header** – 8-bit value where the information about the extension header following IPv6 header is stored. For example, TCP (Transmission Control Protocol) has decimal value 6 and UDP (User Datagram Protocol) value 17.

In case extension headers are used, there can be for example Fragment Header (value 44), ICMPv6 (value 58) and others.

- **Hop Limit** – 8-bit value that limits the number of hops (routers along the path) a packet can travel through. Value gets decremented by 1 with each passage through a router. If the value reaches 0, the packet is discarded and the originator is notified with particular ICMP message.
- **Source Address** – 128-bit address uniquely identifying the originator of a message.
- **Destination Address** – 128-bit address uniquely identifying the destination of a message [17], [18].

In case the extension headers are used, they are placed before the payload part between the standard IPv6 header and the header of a transport layer protocol. This is notified with setting a particular value inside the Next Header item. If such extensions are used, each of them contains the Next Header item as well, which leads to the fact that the extension headers can be chained. The last one carries value of a transport protocol. Extension headers are examined only by the destination node. The only exception is the Hop-by-Hop<sup>2</sup> Options header (value 0), which must be processed by each node the packet travels through. As each extension header has assigned value, the order of chaining must follow these values from the lowest to the highest. This ordering is also done in a way that headers that are processed by the intermediate nodes are put first [17], [18].

## 3.2 IPv4 and IPv6 datagram differences

One of the main differences is that the IPv6 header has constant size of 40 B and the rest is contained inside payload. Compared to that, IPv4 has variable length, where 20 B are fixed and additional parameters are optional. The overall size, however, must be a multiple of 4 B (the maximum is 65 535 B). The source and destination addresses remain the same, but their size is (as discussed earlier) different. The version of protocol is also preserved. Some items have the same meaning for protocol, but different name. An example of such change is Traffic Class and ToS (Type of Service) fields. They both serve the same purpose, which is QoS ensuring. The other equivalents are the Next Header Protocol (Protocol field in IPv4) and the Hop Limit (TTL field in IPv4). One comparison, not so similar, is the Payload Length in IPv6 and Total Length in IPv4. As the name suggests, IPv6 transfers only the information about the payload size as the basic header length is constant,

---

<sup>2</sup>The so called "jumbogram" can be created. It is a packet whose payload size is greater than standard. It can be between 64 kB and 4 GB.



while the IPv4 records the whole length of a packet. Another case is for the Next Header, which can specify additional IPv6 headers as opposed to IPv4, which only specifies the upper layer protocol. Fragmentation related information are moved to the extension header in IPv6, therefore the IPv4 fields Identification, Flags and Fragment Offset are not present in the standard header. Other removed IPv4 fields are the Header Checksum, Options and Padding. Header Checksum is not really necessary as there are mechanism both at the upper and lower layers. The Options field was used for the non-compulsory mechanisms like for example security, route recording and internet timestamp. As not all the options use the same length, Padding is necessary to compensate the total packet length to a multiple of 32 bits [19].

Although IPv6 has an extension header for the fragmentation, it is not really desired to fragment the packets anymore. Originally, IPv4 allowed packets to be fragmented along the path (performed by routers) because of possible different MTU (Maximum Transmission Unit). IPv6 prevents routers from fragmentation, it is possible only at the source. Larger packets than the path MTU are discarded and originator is notified with the ICMP message. Sender can also try to determine optimal MTU along the whole path to the destination by testing the different packet sizes and storing the ICMP "Packet Too Big" messages inside local cache. This process is called the IPv6 MTU Path Discovery. The standard packet size is 1500 B in the Ethernet networks and IPv6 has the minimum packet size of 1280 B [18], [19], [20].

### 3.3 IPv6 addresses

As mentioned many times, IPv6 addresses are 128-bit long. It wouldn't be really efficient to follow the same way as IPv4 addresses are written, that is 4 octets represented in decimal format and divided by dots. IPv6 addresses are represented in a hexadecimal format. There are 8 groups of 4 hexadecimal numbers divided by colon. Network mask for IPv6 follows the prefix notation used in IPv4 (number after a slash represents continuous range of bits set to value 1). There are certain rules that allow to abbreviate the notation:

1. The longest continuous sequence of zeros can be shortened with two colons. This can be used only once in the notation, otherwise ambiguity would arise.
2. Each group of four hexadecimal numbers can be abbreviated by omitting the leading zeros.

### 3.3.1 Types of addresses

IPv4 has 3 types of addresses:

- **Unicast** – Communication one-to-one. One originator sends data to one specific destination.
- **Multicast** – Communication one-to-many. One originator sends data to multiple destinations identified by a specific group address (an example could be video streaming).
- **Broadcast** – Communication one-to-all. One originator sends data to all the stations belonging to the specific address space.

IPv6 defines similar address types. Unicast and multicast preserve the same functionality as in the case of IPv4. However, broadcast is not defined. For the purpose of sending data to multiple sources, multicast is used. Apparently, broadcast would cause extensive packet flooding across the network and would lead to waste of computational resources. There exist special types of multicast addresses that represent, for example, all hosts or all routers in the network. They are used in certain protocols (e.g. Neighbor Discovery protocol). The third type of address defined in IPv6 is **anycast**. The communication works on the principle one-to-closest. Packet is sent to a specific group, however, only the closest (often physically located as there is the shortest delay) node will receive the packet. This mechanism can be used when users are accessing CDN (Content Delivery Network). Data centers are defined with the same IP address, but they are distributed all around the world. Users are redirected to the closest destination to experience the best performance. Load balancing and availability (redundancy) are ensured using such mechanism. Anycast is often closely bound to DNS (Domain Name System) [18], [21].

### 3.3.2 Subtypes of unicast addresses

Interface running IPv6 can have assigned, unlike in case of IPv4, more than one address, which is also necessary for a client to be able to communicate to the internet. There are following types defined:

- **Unspecified Address** – This type of address consists of all zeros ( $0:0:0:0:0:0:0:0$  or simply  $::$ ) and represents the absence of address. A node has to generate the address itself or get one assigned.
- **Loopback Address** – This address ( $0:0:0:0:0:0:0:1$  or  $::1$ ) is used for the local communication inside a node (packets with this either source or destination address never leave the node).
- **Global Unicast Address** – Represents a unique address across the whole internet. The address space currently reserved is  $2000::/3$  (actually range

*0x2000* to *0x3fff*). It can be seen as an equivalent to the IPv4 public addresses, therefore it is routable via internet. It is composed of the 3 main parts:

- **Global Routing Prefix** – This is an equivalent to the IPv4 network address, 48-bit value assigned by the IANA (Internet Assigned Numbers Authority)<sup>3</sup>. It is desired to accumulate the close address ranges physically nearby to exploit aggregation and reduce routing table entries.
- **Subnet ID** – 16-bit value is reserved for the subnetting. Unlike in IPv4, where subnetting was done in a way that bits were borrowed from the host part, IPv6 has allocated portion for such purpose. Each organisation with assigned address space can create its own globally unique subnets.
- **Interface ID** – 64-bit value to uniquely represent each device (interface). It is also possible to borrow bits and create subnets from this range, but only with local validity.

The address can be configured in 3 ways: manual configuration, automatically generating interface ID to the obtained network prefix (stateless address auto-configuration) or getting the address assigned from a DHCPv6 server (stateful DHCPv6). Most commonly, devices generate their own identifier. This does not apply only for the global address, but also for others (i.e. Link-Local and Unique Local). Device can generate the identifier randomly, which is used for generating temporary and constant addresses. Temporary addresses are usually used for the outgoing connections, constant addresses for the incoming. Another approach is Modified EUI-64 (Extended Unique Identifier) which is a standard defined by the IEEE (Institute of Electrical and Electronics Engineers) organisation. This method uses certain identifier of an interface, e.g. MAC (Media Access Control) address in Ethernet network. MAC address is 48 bits in length, therefore 16 additional bits must be added. The constant value *0xfffe* is inserted between the two halves<sup>4</sup> of the MAC address and 7th bit (called also U bit) is inverted. Final value of this bit corresponds to the global scope if it is set to 1 or the local scope when set to 0 [18], [22], [23], [24].

- **Link-Local Address** – These addresses are valid only within a single link (local scope of subnet). Each device must have one and can use the same Link-Local address across multiple interfaces. They can be used for a communication between devices inside Ethernet network and for the purposes like Neighbor Discovery or communication with DHCPv6 server to obtain the ad-

---

<sup>3</sup>Hierarchically assigned to a RIR (Regional Internet Registry), then a LIR (Local Internet Registry) and finally to the customer.

<sup>4</sup>First half (24 bits) represents OUI (Organizational Unique Identifier) which uniquely identifies the manufacturer of the NIC (Network Interface Controller) and the latter part is a serial number.

dress. Link-Local addresses are always generated by the devices themselves after activating IPv6 protocol. It has a specific structure: first 10 bits have constant value  $111111010b$ , they are followed by 54 zeros and the last 64 bits represent the interface identifier. The address space reserved is  $fe80::/10$  (actually range  $0xfe80$  to  $0xfebf$ ). Packets with these addresses are never forwarded by the routers.

- **Unique Local Address** – This is an equivalent of the private IPv4 addresses. Unique Local addresses are used for the communication only within one network or for the communication among multiple networks that are under the same administration and routing can be secured. These addresses should never reach internet. The address space reserved is  $fc00::/7$  (actually range  $0xfc00$  to  $0xfdff$ ). The address is composed of constant bit string  $1111110b$  followed by the L bit, which should be set to 1 to be valid.
- **IPv4 Embedded Address** – This type of addresses is used to carry an IPv4 address inside an IPv6 address which is one way to make the transition from IPv4 to IPv6 simpler (or ensure their coexistence). The address is made of 80 bits set to zero, followed by the constant  $0xffff$  and finally padded with the IPv4 address. An interesting fact is that first 96 bits are represented in hexadecimal notation while the last 32 bits (the IPv4 address) in dotted-decimal notation. IPv4 address does not have to be unique over the internet [18], [22], [23].

### 3.3.3 Multicast addresses

Multicast addresses are used, in a similar way to IPv4, to identify a group of devices (interfaces) that should receive the same content. Real-time services, such as multimedia content sharing (e.g. video streaming), are a typical example. The address space reserved is  $ff00::/8$ . The structure is defined as a string of 8 bits set to value 1. This sequence is followed by 4 bits representing the flags designated as *ORPT*. The first flag is always set to 0 and for example the T (transient) flag represents the well-known addresses assigned by the IANA, which are permanent (value 0) or temporary addresses, dynamically assigned by the application (value 1). The examples of well-known addresses are  $ff02::1$  representing all the devices on a link with IPv6 enabled and  $ff02::2$  representing only the routers on the link with IPv6. Then, 4 bits expressing the scope come after the flags. There are 16 possible values, some of them unassigned. Value  $0x2$  represents a Link-Local scope,  $0x5$  Site-Local scope and  $0xE$  Global scope. At last, 112 bits define a group identifier. Multicast addresses are used to identify a destination, not a source [18], [22], [23].

## 3.4 Internet Control Message Protocol version 6

ICMPv6 (Internet Control Message Protocol version 6) is a network protocol serving for many purposes such as reporting faulty states, testing reachability of hosts and exchanging service information. ICMPv6 is a basic building stone of IPv6, therefore every node in such networks must support it. This protocol incorporates much more features than its equivalent in IPv4. These could be mechanisms such as Neighbor Discovery, distribution of prefixes for automatic address configuration, reporting to multicast groups etc. The structure of an ICMPv6 message is shown in the Fig. 3.2.

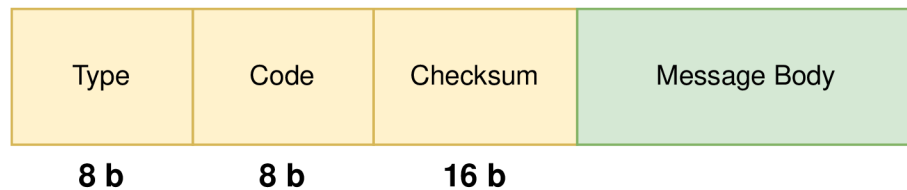


Fig. 3.2: Basic structure of ICMPv6 message.

Following items are present inside ICMPv6 message:

- **Type** – 8-bit field classifies a message and in turn how the remaining items (Message Body) is formatted. Based on the value, ICMPv6 messages can be divided into **error messages** group (values 0–127) and **informational messages** group (values 128–255).
- **Code** – 8-bit value, which indicates specific message inside the particular group of messages based on the type field.
- **Checksum** – 16-bit field to check the correctness of a received message. If the checksum performed by the receiver does not correspond to this value, the packet is dropped and the originator is informed with an error message.
- **Message Body** – The length of this field varies as the content changes according to the Type and Code fields.

The error messages can be further divided into 4 subgroups. The first group, **Destination Unreachable**, is sent by routers that encounter impossibility in delivering packets further to a destination. The reasons might be unknown destination, firewall rules, corrupted following node (either due to address or port) in the path etc., each of which has its own code. The second group, **Packet Too Big**, as the name suggests, informs the source that a packet cannot be delivered due to its size exceeding the MTU of a link. As described earlier, IPv6 routers cannot fragment packets (from other sources) unlike in IPv4. The third group, **Time Exceeded**, there are 2 cases which might occur (identified by their own codes). The first one is when the Hop Limit reaches value 0 and the second one when all the fragments

do not arrive in time to the router. The last group, **Parameter Problem**, designates the problem with processing message at the receiver side. There are 11 codes defined, for example wrong header field (code 0) or unrecognized Next Header type (code 1).

The information messages cover standard use case for testing reachability of nodes. There are 2 types, **Echo Request** and **Echo Reply**. Both messages contain Identifier, Sequence Number and Data fields inside Message Body to uniquely identify which responses belong to particular requests. There are special queries defined in the RFC 4620, which allow sender to obtain information about an address or name of a target node. This resembles DNS functionality, but the intention is not to replace DNS in global scope, but rather provide an alternative to find such information in case DNS is not working properly [18], [25], [26].

## 3.5 Neighbor Discovery for IPv6

ND (Neighbor Discovery) is a protocol defined in the RFC 4861 [27]. It encompasses many functions in IPv6 networks. One of them is the replacement of ARP (Address Resolution Protocol) protocol, which is used to discover a MAC address of a device inside IPv4 Ethernet network. The other purposes are discovering neighbors, routers, updating information (regarding MAC addresses), detection of duplicity addresses and delegating information (using specific type of messages), like prefix and certain flags, to the nodes for the automatic address configuration [18], [26].

### 3.5.1 Neighbor MAC address resolution

If a device is connected to the Ethernet network, detects that a destination belongs to the same network and the MAC address is not known, the **Neighbor Solicitation** message is generated. Its structure is shown in the Fig. 3.3.

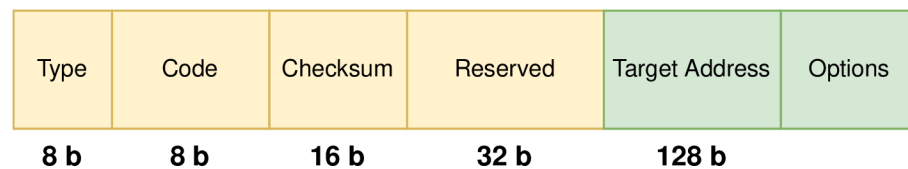


Fig. 3.3: Basic structure of ICMPv6 Neighbor Solicitation message.

- **Type** – 8-bit field set to value 135.
- **Code** – 8-bit field set to value 0.
- **Checksum** – 16-bit value representing standard ICMPv6 checksum.

- **Reserved** – 32-bit value set to all zeros by the sender. This field is not used for useful data and therefore is not analyzed by the receiver.
- **Target Address** – An IPv6 address of a node whose MAC address is desired.
- **Options** – A sender has an option to include his own MAC address inside the *Neighbor Solicitation* message [28].

The *Neighbor Solicitation* message is sent to a special multicast address, called **solicited-node multicast address**, when searching for a MAC address. There is constant prefix *ff02::1:ff00:0/104*, where the last 24 bits are taken from the target IPv6 address. When host simply tests the reachability of a target, unicast address of the target interface is used. The source address is usually an IPv6 address belonging to a particular interface or can be the unspecified address.

When this message is received by a target, it responds with the **Neighbor Advertisement** message. Another use case is when a host wants to inform other nodes about the changes, for example when a new address is configured on an interface and the host sends an update about reachability via its MAC address. The structure is shown in the Fig. 3.4.

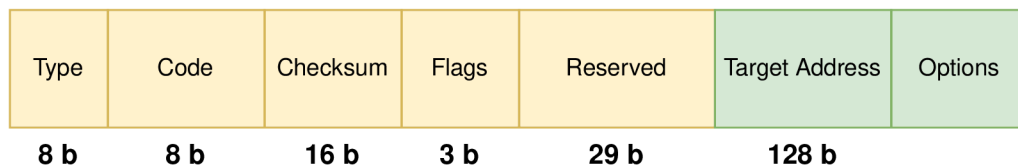


Fig. 3.4: Basic structure of ICMPv6 Neighbor Advertisement message.

- **Type** – 8-bit field set to value 136.
- **Code** – Same as for *Neighbor Solicitation*.
- **Checksum** – Same as for *Neighbor Solicitation*.
- **Flags** – There are 3 flags defined to characterize a message:
  - **R** – The sending host is router if set to value 1.
  - **S** – Solicited flag indicating that the message is sent as a response to the *Neighbor Solicitation*, if set to value 1.
  - **O** – Override flag, where sender should update already existing Neighbor Cache entry and update the link-layer address stored inside cache, if set to value 1.
- **Reserved** – Same as for *Neighbor Solicitation*, but the length is smaller by flags.
- **Target Address** – Remains unchanged when answering the *Neighbor Solicitation*, sets its own IPv6 address when sending an update.
- **Options** – MAC address of a sender [28].

The destination address is either unicast (IPv6 address of the host who sent the *Neighbor Solicitation*) or multicast address **ff02::1** representing all the nodes on the link. The source address is the IPv6 address of a particular interface [26], [27].

## 3.6 Automatic address configuration

There are 2 ways of configuring an IP addresses on the devices. An administrator can assign the address **statically**, which means he selects the address and configures the device accordingly. However, this approach is not sufficient for large networks, where hundreds of clients connect every day and their number might exceed the total amount of available addresses. Currently, static addresses are used for the router's network interfaces, servers, printers and other devices that do not change location and are regularly accessed by the users. Regarding clients, more common way of configuration is **dynamic**. As users change in time, it is appropriate to assign an address temporarily and when a client disconnects, its original address can be used for another. In IPv4 networks, DHCP (Dynamic Host Configuration Protocol) protocol is used for such purpose. The pool of addresses is defined, clients get the addresses assigned mostly temporarily and when they leave, the addresses are returned back to the pool. However, IPv6 comes with more innovative ways of configuring the dynamic IP addresses [18], [26].

### 3.6.1 SLAAC

SLAAC (Stateless Address Autoconfiguration) is a mechanism that allows the address configuration based solely on the communication between a client and a router. Client either contacts a router with the *Router Solicitation* message and as a response, it receives the *Router Advertisement* message, which includes all the necessary parameters (basically flags and prefixes). Another way is that the *Router Advertisement* messages are sent regularly to the network by the routers, therefore there is no necessity to prompt the router [18], [26].

#### Router Solicitation

The structure of a message is very similar to the *Neighbor Solicitation*, however there is no Target Address field and options follow after the reserved part. The Type of the message is set to the value 133 and it is sent to the multicast address covering all the routers on a network (*ff02::2*).



## Router Advertisement

As mentioned before, *Router Advertisement* messages are sent to the network automatically by the routers (regularly based on the set interval), or as a response to the *Neighbor Solicitation* messages. As a response, the message is sent directly to the node which initiated the whole communication. When sent in a predefined interval, the destination is multicast address representing all the nodes on the link (*ff02::1*). The structure is displayed in the Fig. 3.5.

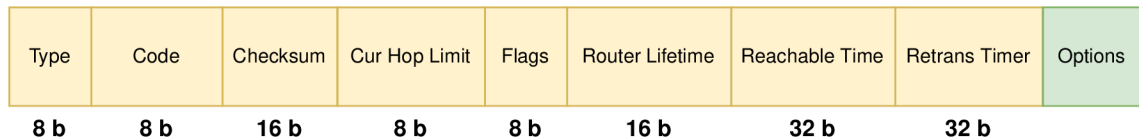


Fig. 3.5: Basic structure of ICMPv6 Router Advertisement message.

It consists of the following fields:

- **Type** – 8-bit field set to value 134.
- **Code** – 8-bit field set to value 0.
- **Checksum** – 16-bit field used to check if there were no errors during the transmission.
- **Cur Hop Limit** – 8-bit field indicating what value should be set in the IPv6 header (Hop Limit field) to limit the number of routers packet can travel through.
- **Flags** – 8-bit field consisting of the following flags:
  - **M** – The 1-bit M flag stands for *Managed address configuration* and if set to 1, a client should contact a DHCPv6 server to obtain an address and other configuration parameters.
  - **O** – The 1-bit O flag stands for *Other configuration* and if set to 1, a client should generate an IP address himself and contact a DHCPv6 server to obtain additional information like the DNS server addresses.
  - **H** – The 1-bit H flag stands for *Home agent*. When set to value 1, router acts in the role of Mobile IPv6 home agent on the link.
  - **Prf** – The 2-bit field representing the preference of implicit routers.
  - **P** – The 1-bit P flag stands for *Proxy* and if set to 1, router acts as a proxy in the ND protocol.
  - **Reserved** – 2-bit value set to all zeros (currently unused).
- **Router Lifetime** – 16-bit value representing the time in seconds, how long the router will be implicit in the given network.

- **Reachable Time** – 32-bit value representing the time in milliseconds, which indicates how long the sender can be assumed as reachable after testing the reachability.
- **Retrans Timer** – Similarly to the Router Lifetime, 32-bit value in milliseconds, which represents time between 2 *Neighbor Solicitation* messages.
- **Options** – In the Options field, useful information for the configuration are carried. There can be MAC address, MTU and most importantly, prefixes representing the network portion of the address to be used. An option to distribute addresses of DNS servers was newly added.

When a client receives the *Neighbor Advertisement* message, it can possess all the useful information to configure the necessary parameters (it generates its host identifier to the given prefix), or based on the flags (M, O) contact the DHCPv6 server for the whole configuration or just additional parameters [18], [26].

### 3.6.2 DHCPv6

As discussed earlier, dynamic configuration is essential in today's networks. It is both convenient and effective to use DHCP servers for the IP address (and other necessary parameters) assignment. In IPv4 networks, clients do not have other option (omitting static configuration and special address ranges like APIPA) to receive an IP address than contacting a DHCP server and leasing one. The communication is viable by unicast (mostly from the server side) or broadcast (mostly from the client side). Until the lease, they have unspecified address and their unique identification is based solely on the physical address. In the IPv6 networks, the situation is quite different. Users have ways to generate their own addresses and the initial communication is with router from which all the necessary details are obtained. Clients can obtain full configuration without being in touch with a DHCPv6 server at all. This depends on the flags in the *Router Advertisement* message. As IPv6 does not support broadcast, special multicast addresses are reserved for the purpose of DHCP communication. These are *ff02::1:2* representing all the DHCP relay agents on the local link and *ff05::1:3* representing all the DHCPv6 servers. Regarding the unique identifier of a client, IPv6 does not rely on MAC addresses, but uses special identifiers called DUID (DHCP Unique Identifier) instead. These should be unique across all the clients and servers and should not change in time. There are more ways of creating these identifiers. This can be left on the manufacturers, where part of the ID is made of the number representing the manufacturer, and the latter part consists of the assigned number specific to that device. Another way is to combine the Link-Local address with the timestamp of its creation and its consecutive storage. The last method is to use the Link-Local address itself. DUID

uniquely defines client, but various interfaces are distinguished by the IA (Identity Association). These identifiers are necessary when DHCP configuration should be obtained for more interfaces on the same client [18], [26].

### Stateful configuration

In a similar way to IPv4 networks, clients obtain the whole configuration from a DHCPv6 server. If the M flag inside *Router Advertisement* message is set to binary value 1, the client knows it should approach the DHCPv6 server to receive all the parameters. There are 4 messages exchanged to fulfill the standard way of communication<sup>5</sup> – SOLICIT, ADVERTISE, REQUEST, REPLY. A client sends the SOLICIT message containing its DUID and IAs. It uses its Link-Local address as the source and destination address of the DHCP relay agents (*ff02::1:2*). When a server (or more servers) receive the message, they reply with the ADVERTISE, where the configuration parameters like IP address, prefix etc. are specified. If more servers are present, the messages may carry also the information about the server preference. Both source and destination addresses are unicast (Link-Local in case of the client, but can be also in case of the server). The client receives all the responses, selects one and answers with the REQUEST message, indicating that he wishes to use this specific configuration. This is sent again to the DHCP relay agents multicast address, but specific server is identified by its DUID. If the server accepts this assignment, it informs the client using the REPLY message. After all these steps, the client can communicate using the given parameters [26].

### Stateless configuration

When the client receives a *Router Advertisement* message carrying flags M set to 0 and O set to 1, this indicates that the address should be auto-configured using one of the available mechanisms to generate the host identifier. For other information, like the DNS server addresses, he should contact appropriate DHCPv6 server. The communication is similar, except that only 2 messages, REQUEST and REPLY, are transmitted [26].

## 3.7 Multicast Listener Discovery

MLD (Multicast Listener Discovery) is the IPv6 protocol used by the clients to report membership to the specific multicast groups for the routers. In order for multicast routers to function correctly, they have to know if there are any clients

---

<sup>5</sup>This is the same for IPv4 networks, but there are slight changes in names of the messages.

listening to the specific multicast traffic so they do not overload the network with an unnecessary traffic. The equivalent protocol in IPv4 networks is IGMP (Internet Group Management Protocol). There are 2 versions of MLD protocol, MLDv1 and MLDv2, where MLDv1 is considered to be an equivalent to IGMPv2 and MLDv2 is an equivalent to IGMPv3. Both versions of MLD protocol use 3 types of messages, **Query**, **Report** and **Done**, but their numbers differ. The Next Header value inside ICMPv6 header is set to value 58, devices use Link-Local addresses as the source address and the Hop Limit is set to value 1, so the packets do not travel beyond the reach of 1 router [26], [28].

### 3.7.1 MLDv1

MLDv1 is more simple protocol in a way that clients only report their membership in specific groups. There is no way of further specifying options like filtering only certain sources. When client enters a multicast group, it sends a packet with the destination address of the group. On the contrary, when a client leaves a group, it sends a packet with the destination of all routers address ( $ff02::2$ ). The packet structure is shown in the Fig. 3.6.

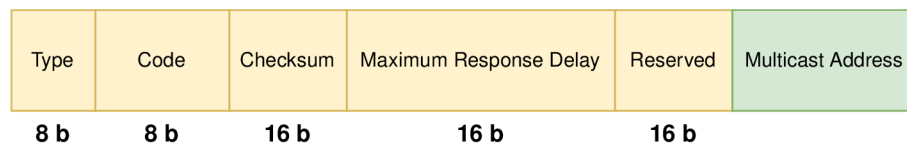


Fig. 3.6: Basic structure of MLDv1 packet.

There are following items:

- **Type** – 8-bit item. As mentioned in the introduction to the MLD protocol, there are 3 types. The **Query** message used by the routers has value 130 and there are 2 different query messages. *General Query* is used to discover clients listening to any multicast addresses (sent to multicast address  $ff02::1$ ) and *Multicast-Address-Specific Query* is used to query specific multicast address, if it has any listeners (sent to multicast address of the group). Then **Report** message with value 131 is used by the clients to report membership in the specific multicast groups. When users leave a group, they send **Done** message with value 132.
- **Code** – 8-bit item set to value 0 by a sender.
- **Checksum** – Standard 16-bit checksum.
- **Maximum Response Delay** – 16-bit item set in the **Query** messages by routers. This is the time in ms, which represents the maximum delay for

the clients to wait before sending the **Report** message as an answer to the **Query**. Each client generates random initial value for the timer. If it receives the **Report** message for the given multicast group from other client, it stops counting and does not send its **Report**. Otherwise, after reaching the *Maximum Response Delay*, it sends its **Report**. The timer is set for each individual multicast address.

- **Reserved** – 16-bit item set to 0 by a sender.
- **Multicast Address** – A sender of either of the messages specifies here the multicast address of interest. The only exception is *General Query* message sent by a router, where the value is set to 0 [28].

### 3.7.2 MLDv2

MLDv2 is more advanced protocol in a way that it enables a client to specify the sources from which the multicast traffic is desired to be received, or, on the other hand, the sources that the client does not wish to receive from. The protocol is also defined in the new RFC (Request for Comments) document 3810 [29], which updates original MLD in RFC 2710. The filtering can be done in 2 ways. The **INCLUDE** filter works in a way that only the addresses mentioned in the record will be accepted as the source point of communication for the given multicast, others will be rejected. In the **EXCLUDE** filter, the multicast communication will be accepted from all the sources except for the ones specified in the record. These filters are sent in the *Report* messages to the *ff02::16* address, which represents all the routers supporting MLDv2. The structure of *Report* packet is displayed in the Fig. 3.7 [26].

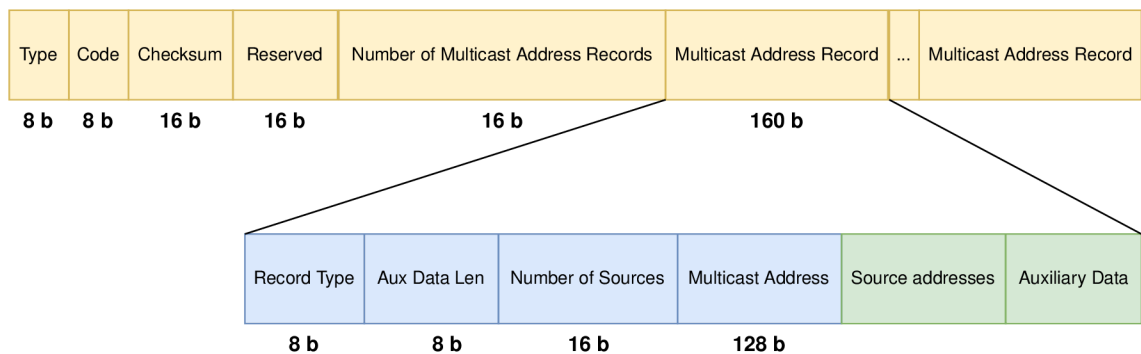


Fig. 3.7: Basic structure of MLDv2 Report packet.

There are items items with similar meaning as in MLDv1 (yellow boxes) and items specific for each filter record (blue boxes). Then follow addresses for filtering. The meaning of each item is described bellow:

- **Type** – 8-bit field set to value 143.
- **Code** – 8-bit field set to value 0.
- **Checksum** – Standard 16-bit checksum.
- **Reserved** – 16-bit field set to value 0.
- **Number of Multicast Address Records** – 16-bit field representing the number of Multicast Address Records (represented by blue + green fields) that are included in the message.
- **Record Type** – 8-bit field representing the type of a filter. The possible values are:
  - `MODE_IS_INCLUDE` (1)
  - `MODE_IS_EXCLUDE` (2)
  - `CHANGE_TO_INCLUDE` (3)
  - `CHANGE_TO_EXCLUDE` (4)
  - `ALLOW_NEW_SOURCES` (5)
  - `BLOCK_OLD_SOURCES` (6)
- **Auxiliary Data Length** – 8-bit field that contains information about the number of auxiliary data (32-bit words) added to this Multicast Address Record.
- **Number of Sources** – 16-bit field giving information about the number of source addresses carried in this record.
- **Multicast Address** – 128-bit field containing the multicast address the record is concerning.
- **Source Addresses** – List of source addresses concerned in this record, whose number corresponds to the Number of Sources field.
- **Auxiliary Data** – Additional data about the record [29].

There are 2 typical cases of using filters for clients. When a client joins a new multicast group, he sets the record type to `MODE_IS_EXCLUDE` without specifying any source addresses, therefore it will listen to all the sources. When the client is leaving a group, he sets the record type to `CHANGE_TO_INCLUDE` without specifying any source addresses, therefore it will not accept any source. **Query** messages are similar to those of MLDv1, but router has options to set specific multicast group addresses together with the sources the clients are supposed to listen to for specific query [26].

## 3.8 IPv6 in operating systems

It was already discussed that IPv4 has insufficient address space according to the needs of the today's world. As the number of users and devices grows larger and wireless sensor networks experience great development, there is need to come with

new addressing scheme as IPv4 addresses were already depleted in 2011. IPv6 solves this problem and it is assumed that the address space of approximately 340 undecillion addresses will be large enough, even with regards to the far future.

Large companies (i.e. Google, Meta, Netflix etc.) started offering their services via IPv6, therefore it was no longer possible for ISP (Internet Service Provider) companies to disregard this new protocol and they had to adapt their infrastructures accordingly. Not only the intermediate devices across the internet must support IPv6, but it is also necessary for the end devices. This responsibility falls on the companies developing operating systems. Probably the most well known enterprises are Microsoft (Windows operating system), Apple (macOS operating system) and various distributions of Linux operating system developed by different organisations (i.e. Red Hat, Canonical, SUSE or even developers all around the world as vast range of distributions is open-source) [30].

### 3.8.1 Linux

Most of the Linux distributions support IPv6 protocol in a default state after the installation, therefore there is no need to activate it manually. To verify the state, `ifconfig` command can be used. It displays all the available interfaces with their corresponding addresses. For the standard network interfaces, Link-Local address could be checked, because this is the only mandatory address an interface should have. Another way is to check the loopback interface, which should have the address `::1/128` assigned. This way IPv6 protocol support may be verified. In case none of these is available, it is possible that the Linux kernel does not support IPv6, even though it is very unlikely today, or it might be necessary to edit the kernel configuration and finally build and install it with necessary modules. The practical commands to perform static/dynamic configuration of the IPv6 addresses are demonstrated in the Numerical results chapter [26].

### 3.8.2 Windows

Microsoft has originally implemented support for IPv6 protocol to Windows XP (Service Pack 1) and to Windows Server 2003, however, the protocol had to be manually enabled and configured using certain commands. The later versions of Windows (for example Windows 7, Vista or Windows Server 2008), or even Windows XP with Service Packs higher than 1, had IPv6 with DHCPv6 support enabled by default. The verification of IPv6 configuration can be done in several ways. One of them is using GUI. The other way might be using command in CMD (Command Prompt) `ipconfig /all` to see IPv6 addresses configured on the individual interfaces, or `netsh interface ipv6 show interface` to display all the

interfaces supporting IPv6. To display only the IPv6 addresses across all the interfaces, `netsh interface ipv6 show addresses` command can be utilized. The practical commands are demonstrated in the Numerical results chapter [26].

### 3.8.3 macOS

The macOS operating system has IPv6 protocol enabled by default. The configuration can be displayed using the `ifconfig` command or GUI designed in a user friendly fashion. All of the configurations of the IPv6 addresses are very similar to the Linux distributions configuration, because macOS is a Unix-based operating system, however macOS is a proprietary software of the Apple company, while Linux is generally open-source [31], [32].



## 4 Numerical results

The practical part of this thesis is described in the following sections. After creating the GNS3 topology, connectivity was tested using the IPv4 static addresses, then the DHCPv4 server and IPv6 was configured on the router, so that the addresses are automatically distributed / generated by the virtual machines. Developed scripts for verification of the *ptnetinspector* tool are described in the Chap. 4.3. In the upcoming examples, only 4 virtual machines plus router are used for the testing scenario demonstrations: Kali, Windows 10, Windows 11 and Ubuntu, see Fig. 4.1. This is due to clarity and the length of output results. Their addresses are displayed in the Tab. 4.1. The whole topology is much larger, as can be seen in the Fig. 4.2. The VMs for testing scenarios are highlighted with colored squares. This topology is used for the research of operating systems responding to different IPv6 flows and to compare the developed scripts with *ptnetinspector*. The addresses for various operating systems are displayed in the Tab.4.2 and for Windows 10 different builds in the Tab.4.3.

### 4.1 ptnetinspector tool

As the basis of this thesis, networking tool called *ptnetinspector* is used. The application designed for Kali Linux was developed by another student [33]. It is able to scan the network by generating IPv6 messages of various protocols, capture the response and print out all the found devices together with their addresses. Basically 3 modes are possible to run that are also utilized in this thesis:

- **Passive mode** – No messages are generated in this mode, only the traffic going to the network interface is sniffed and further analysed. Outgoing traffic is cut by disabling IPs.
- **Active mode** – Traffic is generated in this mode and the responses together with other traffic are stored for the inspection. Queries for the protocols like MLD, mDNS and LLMNR are sent as well as *Router Solicitation* messages, multicast pings, malicious ping (destination type 128) etc.
- **Aggressive mode** – When running this mode, Kali proclaims itself as the router device and distributes *Router Advertisement* messages with default high priority. Machines in the topology thus redirect the traffic to this fake router.

The *ptnetinspector* tool was assessed running 5 proposed testing scenarios in all 3 supported modes. 3 scripts were further developed to verify the tool from the perspective of resource utilization and correctness of the output results. During the verification of results step, the scripts undergo large-scale testing across various operating systems.

## 4.2 Testing topology

The topology created in GNS3 for the testing purposes is shown in the Fig. 4.1. It consists of 6 devices: Kali Linux (version 2023.3) machine which is used as the source point of communication chain and three clients in the testing array – Windows 10 (22H2), Windows 11 (22H2) and Ubuntu (version 22.04.03). The intermediate devices in the topology are Ethernet switch that connects all the devices on the local scope and the Cisco router. The last node, Cloud, is used to connect the topology to the internet.

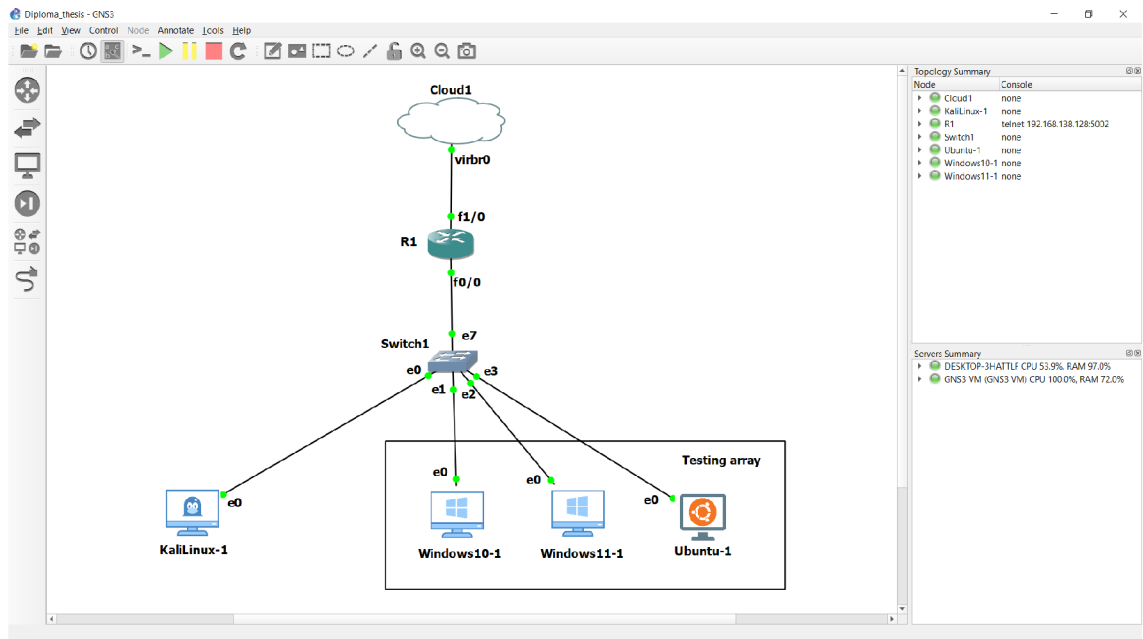


Fig. 4.1: Designed topology in GNS3 for the proposed testing scenarios implementation.

### 4.2.1 Connectivity check

First of all, it is important to check the connectivity whether the devices can communicate with each other. VMs do not have direct access to the internet and there is no DHCP server<sup>1</sup> to automatically assign addresses, therefore manual way of the address configuration is to be performed. Even though GUI can be utilized, CLI (Command Line Interface) way will be demonstrated. The chosen IPv4 address space is 192.168.0.0/24, i.e. private range from the C class. The default gateway address 192.168.0.1/24 is chosen, which will be configured (according to the

<sup>1</sup>The router will be configured later to the role of DHCP server, but this section is used to show the static configuration via command line.

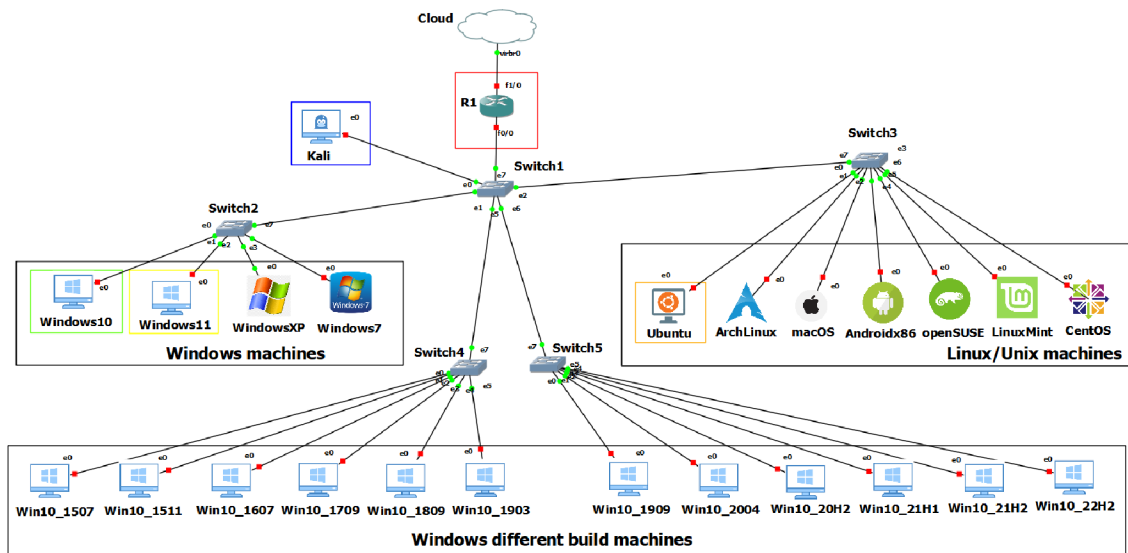


Fig. 4.2: Designed topology in GNS3 for the operating systems IPv6 response re-search.

Table 4.1: GNS3 VM addresses – testing scenarios implementation.

Device	MAC	L-L	GUA
Router	ca01.0906.0000	fe80::c801:9ff:fe06:0	2001:f:b:a::1/64 2001:f:b:a::1/64 2001:f:b:b::1/64 2001:f:b:c::1/64 2001:f:b:d::1/64 2001:f:b:e::1/64
Kali	00:0c:29:b8:a9:4d	fe80::20c:29ff:feb8:a94d	-
Windows 10	00:0c:29:7b:c6:e1	fe80::3ea3:c740:13cb:39a3	2001:f:b:b::2/64
Windows 11	00:0c:29:2a:8e:72	fe80::e734:2b41:5223:b9c9	2001:f:b:b::3/64
Ubuntu	00:0c:29:49:04:db	fe80::f396:bb03:162c:7a59	2001:f:b:b::4/64

topology image) on the router's interface  $f0/0$  (which stands for FastEthernet0/0). Public DNS server 8.8.8.8 is selected. Now the configuration of network interfaces is performed:

- **Kali Linux** – The selected address is 192.168.0.10/24 and is configured in the following way (however this way of configuration is only persistent until the next reboot):

Table 4.2: GNS3 VM addresses – various operating systems.

Device	MAC	L-L	GUA
Windows XP	00:0c:29:65:a1:79	fe80::20c:29ff:fe65:a179	2001:f:b:b::5/64
Windows 7	00:0c:29:0d:49:53	fe80::fc97:2c11:b25:7e19	2001:f:b:b::6/64
Arch Linux	00:0c:29:8d:e4:6d	fe80::3179:3a32:e87c:605c	2001:f:b:b::8/64
macOS	00:0c:29:ba:7f:b2	fe80::1480:8e66:1174:dbd3	2001:f:b:b::7/64
Android	00:0c:29:a8:5e:91	fe80::20c:29ff:fea8:5e91	2001:f:b:b::c/64
openSUSE	00:0c:29:86:f3:b4	fe80::20c:29ff:fe86:f3b4	2001:f:b:b::9/64
Linux Mint	00:0c:29:17:26:fe	fe80::20c:29ff:fe17:26fe	2001:f:b:b::a/64
CentOS	00:0c:29:e8:8d:81	fe80::20c:29ff:fee8:8d81	2001:f:b:b::b/64

```

1 sudo ifconfig eth0 192.168.0.10 netmask 255.255.255.0
2 sudo route add default gw 192.168.0.1
3 sudo sh -c "echo nameserver 8.8.8.8 > /etc/resolv.conf"

```

- **Windows 10&11** – The selected address for Windows 10 is 192.168.0.20/24 and for Windows 11 192.168.0.40/24. As the sequence of commands is the same but with different addresses, the configuration is shown only for the Windows 10 client (command line must be run with administrator privileges):

```

1 netsh interface ipv4 set address name="Ethernet0 2"
   ↪ static 192.168.0.20 255.255.255.0 192.168.0.1
2 netsh interface ipv4 set dns name="Ethernet0 2" static
   ↪ 8.8.8.8

```

- **Ubuntu** – The selected address is 192.168.0.30/24 and is configured in the following way:

```

1 nmcli connection add con-name "ip-static" ifname ens33
   ↪ type ethernet ip4 192.168.0.30/24 gw4 192.168.0.1
2 nmcli connection modify "ip-static" ipv4.dns "8.8.8.8"
3 nmcli connection modify "ip-static" ipv4.method manual
4 nmcli connection modify "ip-static" ipv6.method manual
5 nmcli connection up "ip-static" ifname ens33

```

- **Router** – The default gateway address is 192.168.0.1/24 (interface *f0/0*). The router device is configured using the standard configuration method. The sequence of commands looks as following:

Table 4.3: GNS3 VM addresses – Windows 10 builds.

Device	MAC	L-L	GUA
Windows 10 1507	00:0c:29:0d:b2:e0	fe80::593e:cf0f:944a:b3d4	2001:f:b:b::a1/64
Windows 10 1511	00:0c:29:4a:84:de	fe80::48af:6aa7:e9db:71b9	2001:f:b:b::a2/64
Windows 10 1607	00:0c:29:1c:68:18	fe80::fc9c:7e8b:8b5e:8f8b	2001:f:b:b::a3/64
Windows 10 1709	00:0c:29:98:92:7a	fe80::297f:f2a9:19c5:b5d4	2001:f:b:b::a4/64
Windows 10 1809	00:0c:29:ab:0f:fa	fe80::c517:46a5:862a:f730	2001:f:b:b::a5/64
Windows 10 1903	00:0c:29:8d:ec:27	fe80::4145:cec1:9644:b13e	2001:f:b:b::a6/64
Windows 10 1909	00:0c:29:56:c7:99	fe80::2827:affc:c12b:b8c9	2001:f:b:b::a7/64
Windows 10 2004	00:0c:29:b0:16:cd	fe80::89ea:822c:3f2c:a7e	2001:f:b:b::a8/64
Windows 10 20H2	00:0c:29:35:b9:9e	fe80::a0f9:8849:799d:b269	2001:f:b:b::a9/64
Windows 10 21H1	00:0c:29:54:9c:36	fe80::ed58:d529:28ba:173b	2001:f:b:b::aa/64
Windows 10 21H2	00:0c:29:18:c7:87	fe80::4563:c33d:559b:7004	2001:f:b:b::ab/64
Windows 10 22H2	00:0c:29:4e:8c:88	fe80::a0e0:c25a:14a4:642e	2001:f:b:b::ac/64

```

1 R1#configure terminal
2 R1(config)#interface f0/0
3 R1(config-if)#ip address 192.168.0.1 255.255.255.0
4 R1(config-if)#no shutdown
5 R1(config-if)#exit

```

The router is also connected to the Cloud node (which in this case runs via GNS3 VM whose interface is *virbr0*), therefore the configuration of *f1/0* interface will be done via DHCP.

```

1      R1(config)#interface f1/0
2      R1(config-if)#ip address dhcp
3      R1(config-if)#no shutdown
4      R1(config-if)#exit

```

Until now, the configuration of interfaces was performed, but for the devices to be able to reach the internet, NAT mechanism must be also configured on the router. It can be achieved this way:

```

1      R1(config)#interface f0/0
2      R1(config-if)#ip nat inside
3      R1(config-if)#exit
4      R1(config)#interface f1/0
5      R1(config-if)#ip nat outside
6      R1(config-if)#exit
7      R1(config)#access-list 100 permit ip 192.168.0.0
   ↪   0.0.0.255 any
8      R1(config)#ip nat inside source list 100 interface f1/0
   ↪   overload

```

Finally, the gateway of last resort is configured and the configuration saved (so it is not lost after the reset).

```

1      R1(config)#ip route 0.0.0.0 0.0.0.0 192.168.122.1
2      R1(config)#ip domain-lookup
3      R1(config)#ip name-server 8.8.8.8
4      R1(config)#end
5      R1#copy running-config startup-config

```

Arch Linux, openSUSE, Linux Mint and CentOS are configured statically by editing network configuration files inside system specific network directories. Finally the macOS is configured using the command:

```

1      sudo ifconfig en0 inet6 2001:f:b:b::7 prefixlen 64 alias

```

To verify the reachability, ping tests are issued from the Kali machine. The output is displayed in the Fig. 4.3. It can be seen that all the hosts are available on the network. Capturing was also started on the link connecting Kali with switch. The results are displayed in the Fig. 4.4.

Depending on the computational resources of the physical machine (where GNS3 is running), one might notice random packet drop rate during connectivity testing.



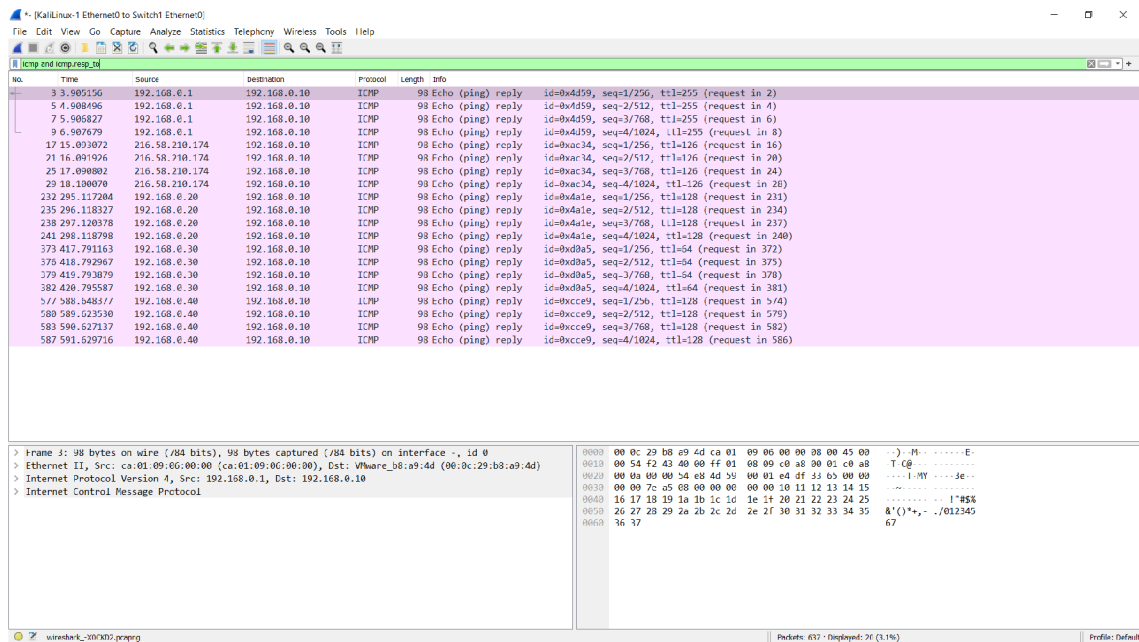


Fig. 4.4: Captured packets of Kali ping in Wireshark.

ICMP traffic designated as ICMPv4-In<sup>2</sup>. The same applies for the IPv6 protocol (ICMPv6-In).

## 4.2.2 IPv4 dynamic configuration

First, the IPv4 configuration is performed. DHCP server is configured on the router and VMs are configured as the DHCP clients to obtain the addresses automatically. This can be done in the following way:

- **Kali Linux** – To set Kali as the DHCP client, the following command can be executed:

```
1 sudo sh -c "echo '\n# The primary network interface\nauto
  ↪ eth0\niface eth0 inet dhcp'
  ↪ >>/etc/network/interfaces"
```

- **Ubuntu** – To set Ubuntu as a DHCP client, the profile is created using the *nmcli* utility. It can be achieved by the following command sequence:

```
1 nmcli connection add con-name "ip-dynamic" ifname ens33
  ↪ type ethernet
2 nmcli connection modify "ip-dynamic" ipv4.addresses ''
3 nmcli connection modify "ip-dynamic" ipv4.gateway ''
```

<sup>2</sup>Beware of choosing the correct profile: either public, private, or domain.



```

4 nmcli connection modify "ip-dynamic" ipv4.dns ''
5 nmcli connection modify "ip-dynamic" ipv4.dhcp-timeout 15
6 nmcli connection modify "ip-dynamic" ipv4.may-fail no
7 nmcli connection modify "ip-dynamic"
  ↪ connection.autoconnect-retries 3
8 nmcli connection modify "ip-dynamic" ipv4.method auto
9 nmcli connection down "ip-static" ifname ens33
10 nmcli connection up "ip-dynamic" ifname ens33

```

- **Windows 10&11** – To set Windows 10 and Windows 11 as the DHCP clients, the following commands can be executed (command line must be run with administrator privileges):

```

1 netsh interface ipv4 set address name="Ethernet0 2" dhcp
2 netsh interface ipv4 set dns name="Ethernet0 2"
  ↪ source=dhcp

```

- **Router** – The router must be configured to act as a DHCP server. The device is configured using the standard configuration method in the following way:

```

1 R1(config)#ip dhcp excluded-address 192.168.0.1
2 R1(config)#ip dhcp pool LAN_VMs
3 R1(dhcp-config)#network 192.168.0.0 255.255.255.0
4 R1(dhcp-config)#default-router 192.168.0.1
5 R1(dhcp-config)#dns-server 8.8.8.8
6 R1(dhcp-config)#lease 1
7 R1(dhcp-config)#end
8 R1#copy running-config startup-config

```

To check the assigned addresses, the following command can be run:

```

1 R1#show ip dhcp binding

```

The possible output is displayed in the Fig. 4.5.

### 4.2.3 IPv6 SLAAC configuration

Next, the IPv6 dynamic configuration using SLAAC mechanism is shown. First of all, router is configured in a way such that it sends RA messages containing prefix periodically to the hosts, which in turn create their global unicast addresses. No

```

R1#show ip dhcp binding
Bindings from all pools not associated with VRF:
IP address          Client-ID/          Lease expiration    Type
Hardware address/
User name
192.168.0.2         ff29.b8a9.4d00.0100.  Oct 26 2023 08:22 AM  Automatic
012c.cb7b.a200.0c29.
b8a9.4d
192.168.0.3         0100.0c29.7bc6.e1    Oct 26 2023 07:14 AM  Automatic
192.168.0.4         0100.0c29.2a8e.72    Oct 26 2023 07:25 AM  Automatic
192.168.0.5         0100.0c29.4904.db    Oct 26 2023 07:58 AM  Automatic
R1#

```

Fig. 4.5: R1 DHCP binding.

additional flags will be set, therefore other details (e.g. domain name or DNS server) will not be obtained<sup>3</sup>. It can be achieved this way:

```

1      R1(config)#ipv6 unicast-routing
2      R1(config)#interface fastEthernet0/0
3      R1(config-if)#ipv6 enable
4      R1(config-if)#ipv6 address 2001:F:B:A::1/64
5      R1(config-if)#ipv6 nd prefix 2001:F:B:A::/64
6      R1(config-if)#ipv6 nd ra interval 10
7      R1(config-if)#end
8      R1#copy running-config startup-config

```

With current configuration, there is no necessity for additional commands, all the VMs will automatically generate their global unicast addresses based on the provided prefix.

## 4.3 Scripts

In this section, the created scripts for testing purposes are explained. Each script will be briefly introduced from the code perspective along with its flowchart. The scripts are available as a part of the Appendix D (or alternatively available on the public GitHub project).

### 4.3.1 Performance testing script

The first script is testing performance of the program from the point of view of the runtime, CPU usage and the RAM usage. The program needs to be run with

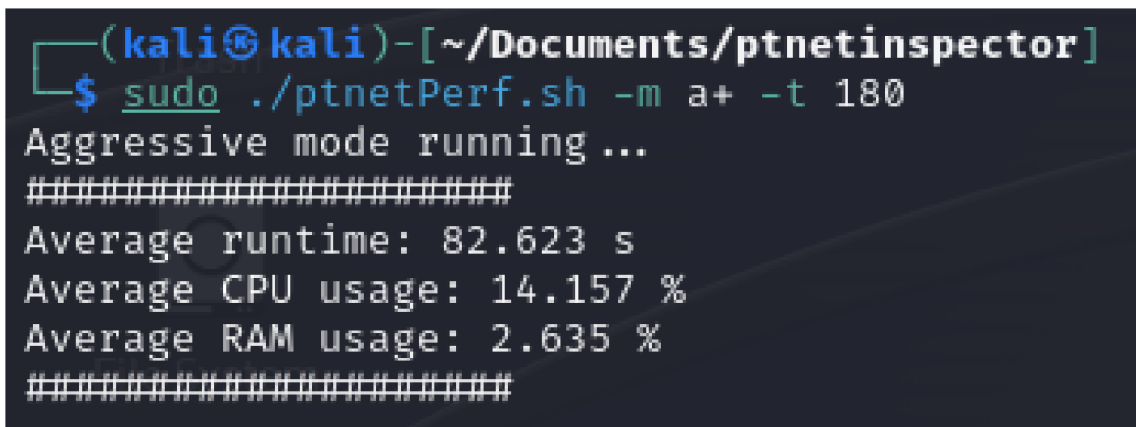
<sup>3</sup>These details would be otherwise received from the DHCPv6 server.

2 flags, the `-m` flag is used for mode selection (`p` for the passive mode, `a` for the active mode and `a+` for the aggressive mode) and the `-t` flag is used for the time value in seconds that the `top` utility will be checked for the program performance. The third (optional) flag, `-r`, serves for the purpose of setting the number of runs. There are mechanisms implemented ensuring the program must be run with the correct flags and correct values. An example of running the script is shown below, the active mode, the measuring time 120 seconds and the number of 5 runs is chosen:

1

```
sudo ./ptnetPerf.sh -m a -t 120 -r 5
```

The flowchart is displayed in the Fig. 4.7. The script created for the purpose of measuring resource utilization runs the program three times by default<sup>4</sup> and captures the statistics from the `time` and `top` utilities. The time is measured in ms and the CPU usage shows current usage of 1 CPU core available. After series of tests, it can be seen that the program briefly utilizes more than 1 core (which makes the utilization over 100 % during sending all of the messages) and then listens to the responses, which does not consume much resources. The longer the program runs, the lower the average value as the peaks are observable only at the beginning. Kali has available 2 cores and 4 GB of RAM in the current VM configuration. As the system has actually 3.89 GB of RAM available, that makes around 100 MB of usage for program. An example of output for the proposed scenario 5 in the active mode is shown in the Fig. 4.6. An example of code is shown in the Appendix B, the rest of the code is present in the `ptnetPerf.sh` file (see Appendix D, or alternatively available on the public GitHub project).



```
(kali@kali)-[~/Documents/ptnetinspector]
└─$ sudo ./ptnetPerf.sh -m a+ -t 180
Aggressive mode running...
#####
Average runtime: 82.623 s
Average CPU usage: 14.157 %
Average RAM usage: 2.635 %
#####
```

Fig. 4.6: Active mode scenario 5 – performance test script output.

---

<sup>4</sup>Can be changed using the `-r` flag.

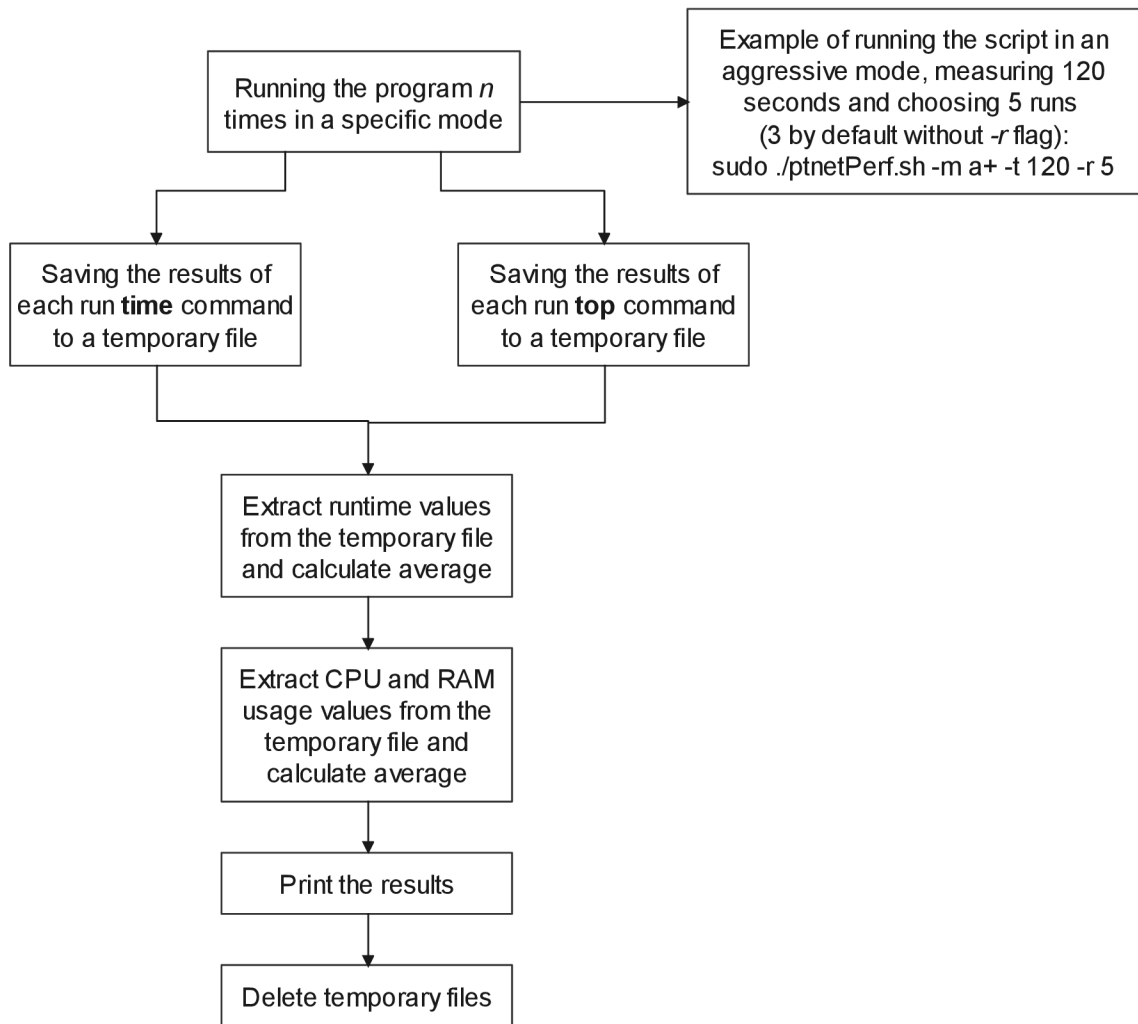


Fig. 4.7: Designed performance testing script flowchart.

### 4.3.2 Traffic capturing script

The script made for running the *ptnetinspector* and capturing the traffic is called as the subprocess from the *Verification of results script* (see the 4.3.3 section). It starts by creating the folder where all the results are then stored. Basic command to run the *ptnetinspector* with the selected mode (passive/active/aggressive) and interface is created. Then additional parameters are added according to the mode. The *tcpdump* process that stores its output in the PCAPNG file and standard error in another file is started in the background. The process ID is stored in a separate variable. By calling the command for the *ptnetinspector*, another process is run in the background and the standard output and standard error are redirected to separate files. The process ID is again stored. When the *ptnetinspector* process finishes, the *tcpdump* process is killed with sending the SIGINT signal to the process.

Artificial delay is added so that the PCAPNG file has time to close correctly<sup>5</sup>. The PCAPNG file is then read and all the packets (except the ones with source MAC address corresponding to the interface where *tcpdump* was running), MLD, mDNS and LLMNR packets are extracted to separate files as well as the standard error. The script contains several messages that inform a user about the current stage of the script for the debug purposes. The flowchart of a script is shown in the Fig. 4.8. An example of code is shown in the Appendix C, the rest of the code is present in the *captPackets.sh* file (see Appendix D, or alternatively available on the public GitHub project).

### 4.3.3 Verification of results script

This script is used for analyzing the results of the *Traffic capturing script* and then compare its results with the *ptnetinspector*.

The code is split into multiple parts (see the Fig. 4.9). At the top, it contains so called Shebang so that the script can be run directly without necessity to specify the interpreter when running from the console. Current interpreter is Python 3.11.5.

The next section contains classes. For analyzing the files created by the *Traffic capturing script*, OOD (Object-Oriented Design) is used. Each unique device identified by its MAC address has 2 lists. The *ip\_addresses* list contains all the captured addresses and those extracted from the payload of mDNS, LLMNR packets etc. The *possible\_ip\_addresses* list contains addresses extracted from the MLD messages. The addresses are either replaced by its full representation (if captured during the communication) or edited and filled with X symbols so that it matches the results of *ptnetinspector*. As all the parameters are private, getters and setters are created.

Following is the global variables section. Here the variables used across the whole code are defined. Specifically, these are 2 dictionaries used for storing the results of *ptnetinspector* and *Traffic capturing script*. The key is represented by the MAC address and value by the Device object described in the classes section. Then there is a list of addresses containing addresses ignored during the analysis and the list storing the start time and end time of the *ptnetinspector* packet capturing<sup>6</sup>.

The next section is functions. Here all the functions used in the code are defined. These functions are, for example, extract source IP, remove port number from the IPv4, extract addresses from the *ptnetinspector* output etc. One function worth mentioning is the one that extracts addresses from the LLMNR packets (payload).

---

<sup>5</sup>Otherwise there were cases where reading failed, specifically when running the passive mode no more than 10 seconds.

<sup>6</sup>The times are acquired from the temporary file using a function.

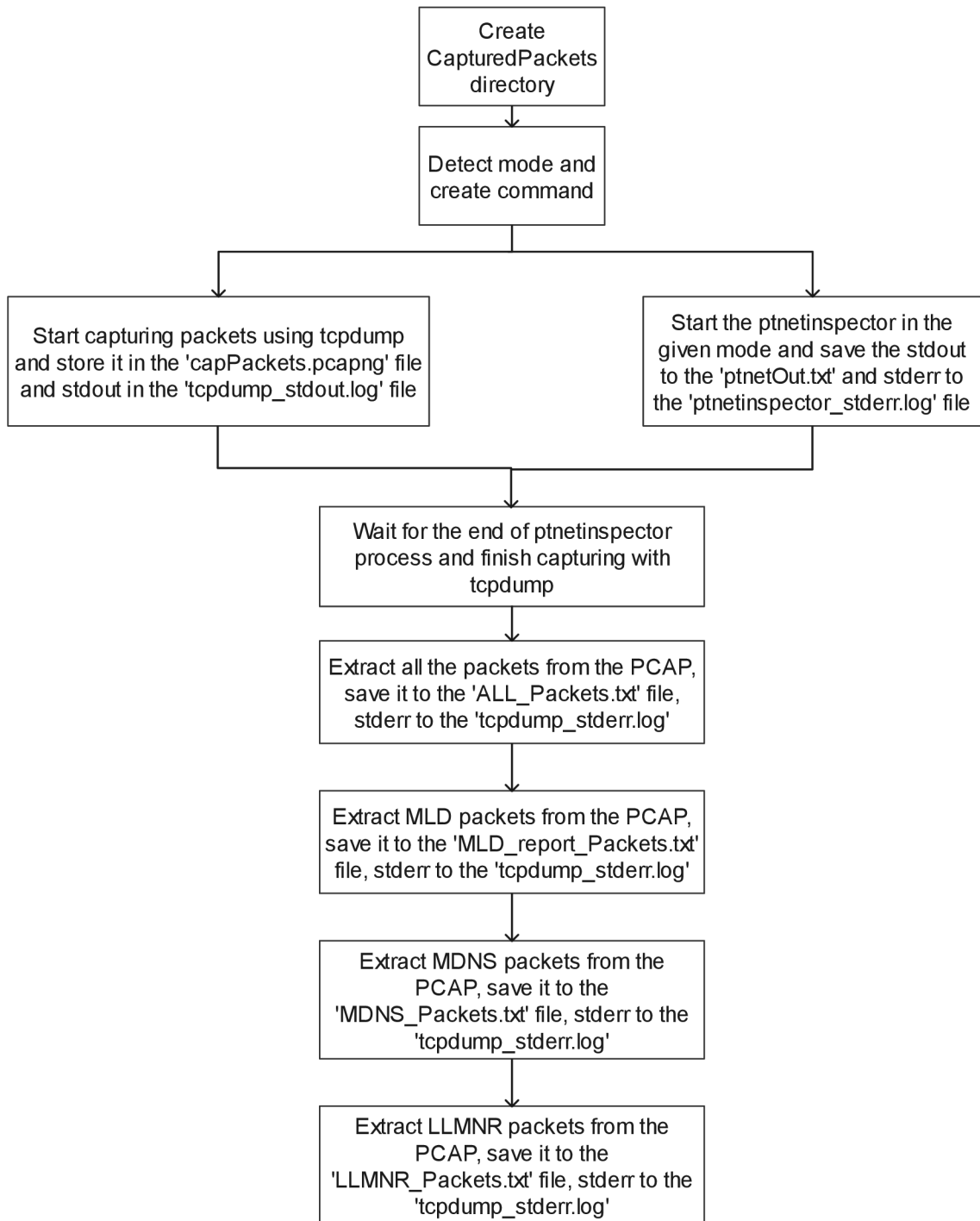


Fig. 4.8: Designed traffic capturing script flowchart.

This was not an easy task as the addresses are not stored in the readable format with repetitive pattern, but individual bytes must be analyzed. The possible cases are A, AAAA and PTR records. PTR records are skipped in the analysis as their addresses always appear inside A or AAAA records. The whole byte array must

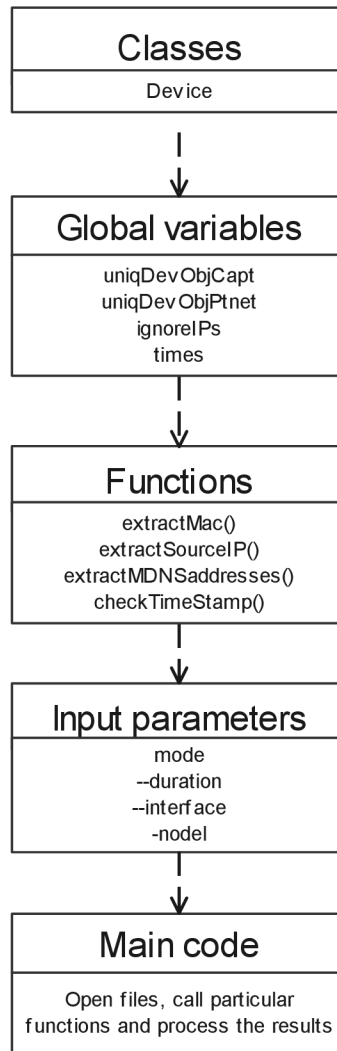


Fig. 4.9: Verification of results script code structure.

be analysed to search for a specific sequence. Then the size of an address and the address itself follow (see Appendix A for the code illustration).

The next section deals with input parameters. In the beginning, the list of all the available network interfaces is created. After that, the flags and other input parameters available from the console are defined. There are compulsory as well as optional arguments. The mode selection is realised with the positional arguments, where further arguments are displayed based on the selected mode. In case that the *node!* argument is set, the subprocess running the Bash script is not called and the analysis is performed using the files from the previous analysis. If the *debug* argument is set, reporting notes are displayed throughout the entire script run, so it is clear which step the script is currently working on. The interface selection is one of the required steps. If the correct interface is selected, its MAC address is automatically extracted. Otherwise a user is warned to select the one from the list

created at the beginning of this part. The help menu can be displayed using the *-h* or *--help* flags.

Then the main code part is defined. In case that the *node1* argument was not given, the mode is analysed and the command structure is created. After that the *Traffic capturing script* is started as subprocess and arguments are relayed to the Bash script. If the script finished correctly and there was no error during the *ptnetinspector* run, all the results are analysed. First, all the packets are analysed to obtain all the individual devices (based on the MAC address) and their respective source IP addresses (either IPv4 or IPv6). After that the *ptnetinspector* captured results are analysed. Following is the analysis of individual IPv6 protocols, each with different approach, except that time synchronization is checked for each packet to avoid distorted results. After all these steps finish, the *possible\_ip\_addresses* set is analysed to either remove already existing addresses or to fill them with ambiguous symbols X.

Finally, the comparison of the two sets is performed. It is a two-way analysis, therefore the *ptnetinspector* results are compared to the *Traffic capturing script* and the other way around so that no address is missed. The results are then stored in the file and displayed in the console.

The flowchart of the script is displayed in the Fig. 4.10. The code is present in the *verifyAddresses.py* file (see Appendix D, or alternatively available on the public GitHub project).



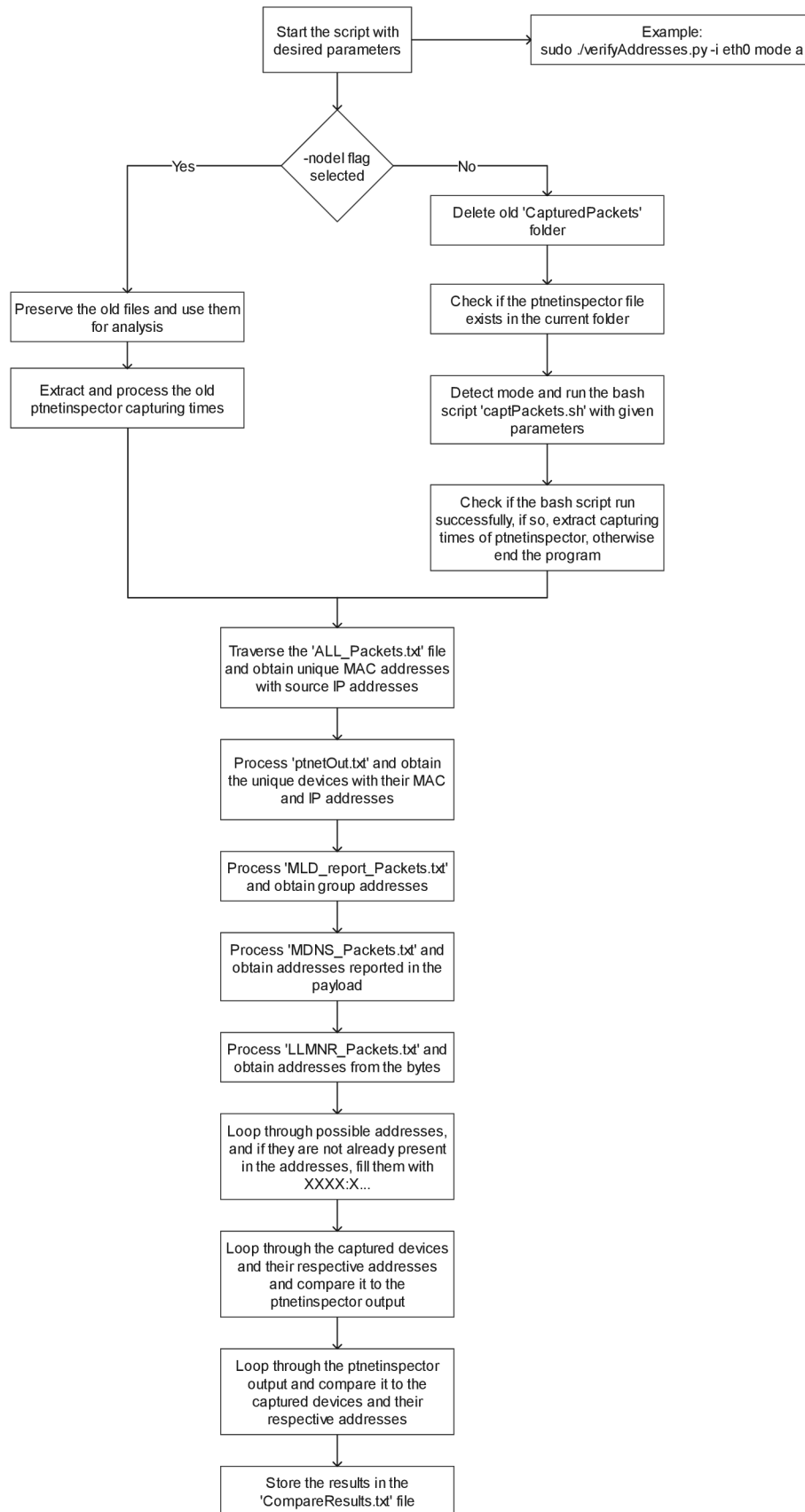


Fig. 4.10: Designed verification of results script flowchart.

## 4.4 Application testing methodology

Out of all the possible testing approaches, the performance testing was selected as the most proper one from the non-functional testing category. The application *ptnetinspector* will undergo testing for various scenarios and the performance will be measured in time. The intended result is to observe the application run time, CPU and RAM utilization, its response to diverse inputs and effectivity in processing results.

### 4.4.1 Testing of Passive mode of the ptnetinspector

In the following points, the testing scenarios are presented. The selected approach is to test the program's response to the increasing input load, which means higher number of addresses used in the network by individual hosts. In the first testing scenario, IPv6 is disabled on the Kali's interface. In the second scenario, passive mode runs when all the stations (except for the router) are successively powered on. In the third scenario, router still remains turned off and stations have configured static global IPv6 addresses. In the fourth scenario, the configuration remains, the router is powered on and delegates 1 IPv6 prefix. In the fifth scenario, router delegates 5 prefixes.

It is important to note that all the hosts have not configured any static address and are trying to reach the DHCP server (IPv4), which is running on the router, or wait for the delegated IPv6 prefix to perform SLAAC, except for Ubuntu. To reach this state on Ubuntu as well, the following commands must be applied to ensure that correct profile is selected:

```
1 nmcli connection modify ip-static connection.autoconnect no
2 nmcli connection modify 'Profile 1' connection.autoconnect no
```

1. **The application will be started after temporarily disabling IPv6 on the interface.**

To disable IPv6 on the *eth0* interface, it is possible to use the following command. This solution is valid only until the next restart (or until the value is set back to 0). If the router, which would immediately start sending RA messages and the operating systems would create their own global address from the distributed prefix, is not started, only the Link-Local address is removed from the interface.

```
1 sudo systemctl -w net.ipv6.conf.eth0.disable_ipv6=1
```

In this case, there is really no need to measure the execution time as one of the input parameters is the duration for which the tool will be eavesdropping on packets. As Kali Linux was the only machine started in the topology, there was no IPv6 communication captured. Another result discovered is that it doesn't really influence the program execution to disable IPv6 in advance.

2. **The eavesdropping begins when the rest of the machines (one after another except for the router) are started.**

To ensure that the Ubuntu machine has only the Link-Local address, the following command must be entered:

```
1 nmcli con modify ip-dynamic ipv6.method link-local
```

The duration was set to 5 minutes so there is enough time for all the machines to boot correctly. The result of testing is displayed in the Fig. 4.11. In this picture and all the following ones, the hosts are highlighted in the colored frames. The program sometimes displays additional host with the MAC address *00:50:56:c0:00:02*, which is the *VMnet* interface over which the physical machine is connected with Kali. It also generates addresses from the received prefixes. The program was able to correctly identify all the machines in the topology both with their IPv4 and IPv6 addresses. The 2 Windows machines can be clearly recognized by their IPv4 address *169.254.X.X/16* as this range is used when the dynamic configuration fails. Ubuntu does not use this range. Each station starts with the *Neighbor Solicitation* messages to discover if there is another station with the same generated Link-Local address in the network already. After that, the *Router Solicitation*, MLD reports and the *Neighbor Advertisement* messages are sent to the network. But there are also other protocols, such as DHCP, mDNS, LLMNR, that were captured.

After initial querying, the regular messages of similar kind are sent from all the machines. These are ICMPv6 messages conveying the *Router Solicitation* and MLD Multicast Listener Report structures as well as mDNS and DHCP protocols.

3. **The same testing as in the scenario 2, but now each of the hosts has static global IPv6 address besides Link-Local address.**

In this case, all the Windows and Linux hosts will have configured static IPv6 addresses. The selected prefix is *2001:f:b:b::/64*. On the Windows machines, this can be achieved by using the following command (with adaptation to the local interface and address):

```
1 netsh interface ipv6 add address "Ethernet0 2"  
  ↪ 2001:f:b:b::2/64
```

```

(kali@kali)-[~/Documents/ptnetinspector]
└─$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
   inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
   inet6 ::1/128 scope host proto kernel_lo
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
   link/ether 00:0c:29:b8:a9:4d brd ff:ff:ff:ff:ff:ff

(kali@kali)-[~/Documents/ptnetinspector]
└─$ sudo python3 ptnetinspector.py -t p -i eth0 -d 300
[sudo] password for kali:
[i] Interface: eth0 exists
[i] Disabling json output
[i] Temporary files are deleted after all
[i] Passive network scan on interface: eth0 for 300.0 seconds

[v] Found 4 devices
[i] Device number 1: (Host)
    MAC 00:0c:29:2a:8e:72
    IPv4 169,254.99,223
    IPv6 fe80::e734:2b41:5223:b9c9
[i] Device number 2: (Host)
    MAC 00:0c:29:49:04:db
    IPv6 fe80::f396:bb03:162c:7a59
[i] Device number 3: (Host)
    MAC 00:0c:29:7b:c6:e1
    IPv4 169.254.198.169
    IPv6 fe80::3ea3:c740:13cb:39a3
[i] Device number 4: (Host)
    MAC 00:50:56:c0:00:02
    IPv4 169.254.52.37
    IPv6 fe80::e07:8494:45d9:9d55
[i] Passive scan ended

(kali@kali)-[~/Documents/ptnetinspector]
└─$

```

Fig. 4.11: Program output of the passive mode scenario 2 – hosts with Link-Local addresses only.

On the Ubuntu machine, this can be achieved by modifying the *ip-dynamic* profile. Another step must be taken to successfully activate the profile, that is changing the method of IPv6 to *manual*. After the router is powered on and starts distributing prefixes, this method is changed back to *auto*.

```

1 nmcli connection modify "ip-dynamic" ipv6.addresses
   ↪ 2001:f:b:b::4/64
2 nmcli con modify ip-dynamic ipv6.method manual

```

Windows 10 will be configured with the address *2001:f:b:b::2/64*, Windows 11 *2001:f:b:b::3/64* and Ubuntu *2001:f:b:b::4/64*.

It can be seen in the Fig. 4.12 that the program was able to recognize all the addresses of the devices. A short portion of packets captured in Wireshark is displayed in the Fig. 4.13. It can be seen that two IPv6 addresses are configured, but static global address is used only when announcing reachability

via specific MAC address.

```
(kali@kali)-[~/Documents/ptnetinspector]
└─$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host proto kernel_lo
        valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 00:0c:29:b8:a9:4d brd ff:ff:ff:ff:ff:ff

(kali@kali)-[~/Documents/ptnetinspector]
└─$ sudo python3 ptnetinspector.py -t p -i eth0 -d 300
[i] Interface: eth0 exists
[i] Disabling json output
[i] Temporary files are deleted after all
[i] Passive network scan on interface: eth0 for 300.0 seconds

[v] Found 4 devices
[i] Device number 1: (Host)
    MAC 00:0c:29:2a:8e:72
    IPv4 169.254.99.223
    IPv6 2001:f:b:b::3
    IPv6 fe80::e734:2b41:5223:b9c9
[i] Device number 2: (Host)
    MAC 00:0c:29:49:04:db
    IPv6 2001:f:b:b::4
    IPv6 fe80::f396:bb03:162c:7a59
[i] Device number 3: (Host)
    MAC 00:0c:29:7b:c6:e1
    IPv4 169.254.198.169
    IPv6 2001:f:b:b::2
    IPv6 fe80::3ea3:c740:13cb:39a3
[i] Device number 4: (Host)
    MAC 00:50:56:c0:00:02
    IPv4 169.254.52.37
    IPv6 fe80::e07:8494:45d9:9d55
[i] Passive scan ended

(kali@kali)-[~/Documents/ptnetinspector]
└─$
```

Fig. 4.12: Program output of the passive mode scenario 3.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	::	ff02::1:ffcb:39a3	ICMPv6	78	Neighbor Solicitation for fe80::3ea3:c740:13cb:39a3
2	0.003781	::	ff02::1:fff0:2	ICMPv6	78	Neighbor Solicitation for 2001:f:b:b::2
3	0.005730	fe80::3ea3:c740:13cb:39a3	ff02::2	ICMPv6	62	Router Solicitation
4	0.006326	fe80::3ea3:c740:13cb:39a3	ff02::16	ICMPv6	110	Multicast Listener Report Message v2
5	0.505225	fe80::3ea3:c740:13cb:39a3	ff02::16	ICMPv6	110	Multicast Listener Report Message v2
6	0.995550	fe80::3ea3:c740:13cb:39a3	ff02::1	ICMPv6	86	Neighbor Advertisement fe80::3ea3:c740:13cb:39a3 (ovr) is at 00:0c:29:7b:c6:e1
7	0.998046	2001:f:b:b::2	ff02::1	ICMPv6	86	Neighbor Advertisement 2001:f:b:b::2 (ovr) is at 00:0c:29:7b:c6:e1
8	3.995592	fe80::3ea3:c740:13cb:39a3	ff02::2	ICMPv6	70	Router Solicitation from 00:0c:29:7b:c6:e1
11	7.989649	fe80::3ea3:c740:13cb:39a3	ff02::2	ICMPv6	70	Router Solicitation from 00:0c:29:7b:c6:e1
14	20.266766	fe80::3ea3:c740:13cb:39a3	ff02::1:2	DHCPv6	157	Solicit XID: 0xd286b9 CID: 000190012cb4bb82000c297bc6e1
15	21.781734	fe80::3ea3:c740:13cb:39a3	ff02::1:2	DHCPv6	157	Solicit XID: 0xd286b9 CID: 000190012cb4bb82000c297bc6e1
17	23.809352	fe80::3ea3:c740:13cb:39a3	ff02::1:2	DHCPv6	157	Solicit XID: 0xd286b9 CID: 000190012cb4bb82000c297bc6e1
19	27.908346	fe80::3ea3:c740:13cb:39a3	ff02::1:2	DHCPv6	157	Solicit XID: 0xd286b9 CID: 000190012cb4bb82000c297bc6e1
20	28.510004	fe80::3ea3:c740:13cb:39a3	ff02::16	ICMPv6	90	Multicast Listener Report Message v2
22	28.528237	fe80::3ea3:c740:13cb:39a3	ff02::16	ICMPv6	90	Multicast Listener Report Message v2
24	28.872488	fe80::3ea3:c740:13cb:39a3	ff02::16	ICMPv6	90	Multicast Listener Report Message v2
26	28.893826	fe80::3ea3:c740:13cb:39a3	ff02::16	ICMPv6	90	Multicast Listener Report Message v2
29	28.988311	fe80::3ea3:c740:13cb:39a3	ff02::16	ICMPv6	110	Multicast Listener Report Message v2

Fig. 4.13: Passive mode scenario 3 – conversation sample in Wireshark.

4. **All the hosts have static global IPv6 addresses and the router delegates 1 prefix for SLAAC.**

First of all, the address  $2001:f:b:b::1/64$  is assigned to the router interface along with  $2001:f:b:a::1/64$ . As all the hosts are configured for dynamic address configuration (including Kali), each of them will also obtain IPv4 address. The output is displayed in the Fig. 4.14. The tool was able to capture all of the available addresses including APIPA (Automatic Private IP Addressing) address for Windows 11 before it was able to obtain the dynamic address from the router. This comes initially from the IGMPv3 membership reports. The reason behind so many IPv6 addresses belonging to 1 device is that for each prefix it creates one permanent and one temporary IPv6 address, which makes 4 addresses altogether plus one statically configured. Once the interface is configured for certain IPv6 address on the router, it automatically starts with delegation of its prefix, therefore 2 prefixes are sent inside the *Router Advertisement* messages. This also leads to the fact that each machine has 3 addresses from the same prefix.

5. **All the hosts have static global IPv6 addresses and the router delegates 5 prefixes for SLAAC.**

The two prefixes are already delegated from the previous configuration, therefore it is sufficient to configure the interface for another 3 IPv6 addresses. The selected ones are  $2001:f:b:c::1/64$ ,  $2001:f:b:d::1/64$  and  $2001:f:b:e::1/64$ .

All 4 devices were successfully discovered with their respective addresses that were used for the communication across network. The output is too long, therefore only one output (from the Ubuntu device) is shown in the Fig. 4.15. Another interesting part is displayed in the Fig. 4.16. Only 2 out of 5 global addresses were captured, but after the packet analysis in Wireshark, only the  $2001:f:b:a::1$  and  $2001:f:b:e::1$  addresses were used for communication, specifically for the *Neighbor Solicitation* messages where the router announced its link-layer address to the target machines.

The IPv6 protocol was again enabled on the Kali after the Passive mode tests. The generated IPv6 addresses are shown in the Fig. 4.17. It can be observed that unlike in case of the Windows and Ubuntu machines, Kali creates only one global address from the given prefix. EUI-64 mechanism instead of random values is used to generate the identifier in the host portion.

## 4.4.2 Testing of Active mode of the ptnetinspector

In this section, 5 scenarios will be presented running the program in the active mode. Unlike in case of passive mode, all the virtual machines can already run, because

```

2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
link/ether 00:0c:29:b8:a9:4d brd ff:ff:ff:ff:ff:ff
inet 192.168.0.2/24 brd 192.168.0.255 scope global dynamic eth0
valid_lft 86179sec preferred_lft 86179sec

(kali@kali)-[~/Documents/ptnetinspector]
└─$ sudo python3 ptnetinspector.py -t p -i eth0 -d 300
[i] Interface: eth0 exists
[i] Disabling json output
[i] Temporary files are deleted after all
[i] Passive network scan on interface: eth0 for 300.0 seconds

[v] Found 5 devices
[i] Device number 1: (Host)
MAC 00:0c:29:2a:8e:72
IPv4 169.254.99.223
IPv4 192.168.0.4
IPv6 2001:f:b:a:21d0:d656:c50a:2a8
IPv6 2001:f:b:a:6cc5:7b0d:8c24:247
IPv6 2001:f:b:b::3
IPv6 2001:f:b:b:6923:f0d2:b8a2:8947
IPv6 2001:f:b:b:6cc5:7b0d:8c24:247
IPv6 fe80::e734:2b41:5223:b9c9
[i] Device number 2: (Host)
MAC 00:0c:29:49:04:db
IPv4 192.168.0.3
IPv6 2001:f:b:a:4bae:9c40:b1f8:dfba
IPv6 2001:f:b:a:5b68:ae2:e3fc:ee73
IPv6 2001:f:b:b::4
IPv6 2001:f:b:b:a35b:8067:9858:babf
IPv6 2001:f:b:b:f88c:b8c1:2156:bffb
IPv6 fe80::f396:bb03:162c:7a59
[i] Device number 3: (Host)
MAC 00:0c:29:7b:c6:e1
IPv4 192.168.0.5
IPv6 2001:f:b:a:14ab:293c:a122:cc97
IPv6 2001:f:b:a:d200:1e15:e46f:1078
IPv6 2001:f:b:b::2
IPv6 2001:f:b:b:14ab:293c:a122:cc97
IPv6 2001:f:b:b:700e:e565:47a9:2cc6
IPv6 fe80::3ea3:c740:13cb:39a3
[i] Device number 4: (Host)
MAC 00:50:56:c0:00:02
IPv4 169.254.52.37
IPv6 2001:f:b:a:98ad:18e:7f41:b414
IPv6 fe80::e07:8494:45d9:9d55
[i] Device number 5: (Preferred router)
MAC ca:01:09:06:00:00
IPv4 192.168.0.1
IPv6 2001:f:b:a::1
IPv6 2001:f:b:b::1
IPv6 fe80::c801:9ff:fe06:0
[i] Passive scan ended

```

Fig. 4.14: Program output of the passive mode scenario 4.

Kali will actively prompt the machines and will listen to the responses. This was not the case for the passive mode, where after the machines have the addresses already assigned, there is not much ongoing communication.

In the first scenario, the program is started when no other device is present on the network (other hosts are turned off). In the second scenario, the devices (except for router) are powered on, but have only Link-Local addresses. In the third scenario, the router remains turned off and hosts are additionally configured with static global addresses. In the fourth scenario, the router is powered on and distributes 1 prefix

```

[i] Device number 2: (Host)
MAC    00:0c:29:49:04:db
IPv4   192.168.0.3
IPv6   2001:f:b:a:5b68:ae2:e3fc:ee73
IPv6   2001:f:b:a:9a57:939b:2575:8e34
IPv6   2001:f:b:b::4
IPv6   2001:f:b:b:2240:1b42:140:85fa
IPv6   2001:f:b:b:f88c:b8c1:2156:bffb
IPv6   2001:f:b:c:15d9:a9ea:be32:5fb5
IPv6   2001:f:b:c:c986:7bad:541b:1524
IPv6   2001:f:b:d:2a4a:2920:a7a8:1d92
IPv6   2001:f:b:d:7532:1027:f84a:8955
IPv6   2001:f:b:e:806d:dc6a:9f4a:5923
IPv6   2001:f:b:e:a961:ccb9:2641:b580
IPv6   fe80::f396:bb03:162c:7a59

```

Fig. 4.15: Program output of the passive mode scenario 5 for Ubuntu.

```

[i] Device number 5: (Preferred router)
MAC    ca:01:09:06:00:00
IPv6   2001:f:b:a::1
IPv6   2001:f:b:e::1
IPv6   fe80::c801:9ff:fe06:0
[i] Passive scan ended

```

Fig. 4.16: Program output of the passive mode scenario 5 for router.

and in the fifth scenario, the router distributes 5 prefixes.

1. **The application will be started when there is no other IPv6 device on the network.**

In this scenario, no device is powered on, or the devices simply do not have the IPv6 addresses assigned. There is no response to the messages sent by the Kali



```

2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 state UP qlen 1000
  inet6 2001:f:b:e:20c:29ff:feb8:a94d/64 scope global dynamic mngtmpaddr proto kernel_ra
        valid_lft 2591998sec preferred_lft 604798sec
  inet6 2001:f:b:d:20c:29ff:feb8:a94d/64 scope global dynamic mngtmpaddr proto kernel_ra
        valid_lft 2591998sec preferred_lft 604798sec
  inet6 2001:f:b:c:20c:29ff:feb8:a94d/64 scope global dynamic mngtmpaddr proto kernel_ra
        valid_lft 2591998sec preferred_lft 604798sec
  inet6 2001:f:b:b:20c:29ff:feb8:a94d/64 scope global dynamic mngtmpaddr proto kernel_ra
        valid_lft 2591998sec preferred_lft 604798sec
  inet6 2001:f:b:a:20c:29ff:feb8:a94d/64 scope global dynamic mngtmpaddr proto kernel_ra
        valid_lft 2591998sec preferred_lft 604798sec
  inet6 fe80::20c:29ff:feb8:a94d/64 scope link proto kernel_ll
        valid_lft forever preferred_lft forever

```

Fig. 4.17: Kali automatically generated IPv6 addresses.

Linux and therefore the test does not take too long (see Tab. 4.4). The output of the program is shown in the Fig. 4.18. As can be seen in the Fig. 4.19, there was totally 13 packets sent, basically the all-node address (*ff02::1*), the all routers address (*ff02::2*) addresses were used, then the LLMNR (Link-Local Multicast Name Resolution) (*ff02::1:3*) and the IPv6 multicast DNS (*ff02::fb*) were also captured. As no response came, no device was detected.

```

(kali@kali)-[~/Documents/ptnetinspector]
└─$ sudo python3 ptnetinspector.py -t a -i eth0
[i] Interface: eth0 exists
[i] Disabling json output
[i] Temporary files are deleted after all
[i] Active network scan on interface: eth0
[i] Add record to ipv6table
[i] Active scan in progress. This may take a while. Just wait for results
[i] Remove record from ipv6table

-----

[v] Found 1 device
[i] Device number 1: (Host)
    MAC    00:50:56:c0:00:02
    IPv6   fe80::e07:8494:45d9:9d55
[i] Active scan ended

(kali@kali)-[~/Documents/ptnetinspector]
└─$

```

Fig. 4.18: Program output of the active mode scenario 1.

2. The application will be started when other devices are on (except for the router) and have only Link-Local address.

As can be seen in the Tab. 4.4, the results are quite comparable to those from the scenario 1. Now there are 3 devices, each with just Link-Local address, therefore there is not much traffic generated. 36 packets were totally

Address A	Address B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A
fe80::20c:29ff:feb8:a94d	fe80::e07:8494:45d9:9d55	2	172 bajty	2	172 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	ff02::1	7	554 bajty	7	554 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	ff02::1:3	1	152 bajty	1	152 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	ff02::2	2	132 bajty	2	132 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	ff02::fb	1	152 bajty	1	152 bajty	0	0 bajty

Fig. 4.19: IPv6 conversations during active scan in the scenario 1.

transferred (see Fig. 4.20). There were answers to the MLD queries from all the hosts, as well as *Neighbor Solicitation* messages when the Kali introduced himself using the *Neighbor Advertisement* message. All of the addresses were correctly captured by the program.

Address A	Address B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A
fe80::20c:29ff:feb8:a94d	fe80::3ea3:c740:13cb:39a3	6	508 bajty	3	250 bajty	3	258 bajty
fe80::20c:29ff:feb8:a94d	fe80::e07:8494:45d9:9d55	2	164 bajty	2	164 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	fe80::e734:2b41:5223:b9c9	6	508 bajty	3	250 bajty	3	258 bajty
fe80::20c:29ff:feb8:a94d	ff02::1	7	554 bajty	7	554 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	ff02::1:3	3	456 bajty	3	456 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	ff02::2	2	132 bajty	2	132 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	ff02::fb	3	456 bajty	3	456 bajty	0	0 bajty
fe80::3ea3:c740:13cb:39a3	ff02::16	2	300 bajty	2	300 bajty	0	0 bajty
fe80::3ea3:c740:13cb:39a3	ff02::1:2	2	314 bajty	2	314 bajty	0	0 bajty
fe80::3ea3:c740:13cb:39a3	ff02::1:ffcb:39a3	1	86 bajty	1	86 bajty	0	0 bajty
fe80::e734:2b41:5223:b9c9	ff02::16	1	150 bajty	1	150 bajty	0	0 bajty
fe80::e734:2b41:5223:b9c9	ff02::1:ff23:b9c9	1	86 bajty	1	86 bajty	0	0 bajty

Fig. 4.20: IPv6 conversations during active scan in the scenario 2.

3. **The application will be started when other devices are on (except for the router) and have the Link-Local and static addresses.**

The output of the program is shown in the Fig. 4.21). It was able to process all of the addresses mainly based on the MLD report messages. The only manually configured address fully recognized belongs to the Ubuntu machine, which sent mDNS query using this address. The conversations are shown in the Fig. 4.22. 76 packets were totally transferred.

```
(kali㉿kali)-[~/Documents/ptnetinspector]
└─$ sudo python3 ptnetinspector.py -t a -i eth0
[i] Interface: eth0 exists
[i] Disabling json output
[i] Temporary files are deleted after all
[i] Active network scan on interface: eth0
[i] Add record to ipv6table
[i] Active scan in progress. This may take a while. Just wait for results
[i] Remove record from ipv6table

-----
[v] Found 4 devices
[i] Device number 1: (Host)
    MAC    00:0c:29:2a:8e:72
    IPv6   fe80::e734:2b41:5223:b9c9
    IPv6   XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XX00:0003 (possible address)
[i] Device number 2: (Host)
    MAC    00:0c:29:49:04:db
    IPv6   2001:f:b:b::4
    IPv6   fe80::f396:bb03:162c:7a59
[i] Device number 3: (Host)
    MAC    00:0c:29:7b:c6:e1
    IPv6   fe80::3ea3:c740:13cb:39a3
    IPv6   XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XX00:0002 (possible address)
[i] Device number 4: (Host)
    MAC    00:50:56:c0:00:02
    IPv6   fe80::e07:8494:45d9:9d55
[i] Active scan ended
```

Fig. 4.21: Program output of the active mode scenario 3.

4. **The application will be started when all the devices are powered on (containing all the previous addresses) and the router propagates 1 prefix.** The output is displayed in the Fig. 4.23. All the IPv6 and IPv4 addresses were successfully captured and assigned to the correct devices. The only devices without IPv4 addresses in the output are the router and the Ubuntu machine. This is correct as they were neither captured by Wireshark. The conversations are shown in the Fig. 4.24. Regarding the performance table results, there were many variations in the execution time of the program. The times were irregular, mostly around 20 seconds or 1 minute.

Address A	Address B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A
2001:fb:b::4	ff02::fb	4	614 bajty	4	614 bajty	0	0 bajty
::	ff02::16	3	350 bajty	3	350 bajty	0	0 bajty
::	ff02::1:ff00:4	1	86 bajty	1	86 bajty	0	0 bajty
::	ff02::1:ff2c:7a59	1	86 bajty	1	86 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	fe80::3ea3:c740:13cb:39a3	2	172 bajty	1	86 bajty	1	86 bajty
fe80::20c:29ff:feb8:a94d	fe80::e07:8494:45d9:9d55	2	164 bajty	2	164 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	fe80::e734:2b41:5223:b9c9	2	172 bajty	1	86 bajty	1	86 bajty
fe80::20c:29ff:feb8:a94d	fe80::f396:bb03:162c:7a59	8	712 bajty	3	258 bajty	5	454 bajty
fe80::20c:29ff:feb8:a94d	ff02::1	7	554 bajty	7	554 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	ff02::1:3	5	760 bajty	5	760 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	ff02::2	2	132 bajty	2	132 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	ff02::fb	5	760 bajty	5	760 bajty	0	0 bajty
fe80::3ea3:c740:13cb:39a3	ff02::1:3	1	86 bajty	1	86 bajty	0	0 bajty
fe80::3ea3:c740:13cb:39a3	ff02::1:ff00:2	4	344 bajty	4	344 bajty	0	0 bajty
fe80::3ea3:c740:13cb:39a3	ff02::1:ffcb:39a3	4	344 bajty	4	344 bajty	0	0 bajty
fe80::3ea3:c740:13cb:39a3	ff02::c	1	86 bajty	1	86 bajty	0	0 bajty
fe80::3ea3:c740:13cb:39a3	ff02::fb	1	86 bajty	1	86 bajty	0	0 bajty
fe80::e734:2b41:5223:b9c9	ff02::1:3	1	86 bajty	1	86 bajty	0	0 bajty
fe80::e734:2b41:5223:b9c9	ff02::1:ff00:3	4	344 bajty	4	344 bajty	0	0 bajty
fe80::e734:2b41:5223:b9c9	ff02::1:ff23:b9c9	4	344 bajty	4	344 bajty	0	0 bajty
fe80::e734:2b41:5223:b9c9	ff02::c	1	86 bajty	1	86 bajty	0	0 bajty
fe80::e734:2b41:5223:b9c9	ff02::fb	1	86 bajty	1	86 bajty	0	0 bajty
fe80::f396:bb03:162c:7a59	ff02::16	4	520 bajty	4	520 bajty	0	0 bajty
fe80::f396:bb03:162c:7a59	ff02::1:ff00:4	2	172 bajty	2	172 bajty	0	0 bajty
fe80::f396:bb03:162c:7a59	ff02::1:ff2c:7a59	2	172 bajty	2	172 bajty	0	0 bajty
fe80::f396:bb03:162c:7a59	ff02::1:ffb8:a94d	1	86 bajty	1	86 bajty	0	0 bajty
fe80::f396:bb03:162c:7a59	ff02::2	3	186 bajty	3	186 bajty	0	0 bajty

Fig. 4.22: IPv6 conversations during active scan in the scenario 3.

5. **The application will be started when all the devices are powered on (containing all the previous addresses) and the router propagates 5 prefixes.**

The program was able to recognize all of the addresses belonging to the network interfaces of other VMs. An example of output is shown in the Fig. 4.25. The communication starts with Kali sending MLDv1 and MLDv2 queries and collecting all the responses, in which VMs report their membership to the multicast groups. Then Kali sends ping request to the *all-nodes address* and either receives the ping response directly (from the Ubuntu machine) or *Neighbor Solicitation* message and answers with *Neighbor Advertisement* including its own MAC address. Router sends *Neighbor Solicitation* message from all the IPv6 global unicast addresses (5 totally) it has on the interface. Program then sends sequences of messages according to its settings (malicious ping, ping to the all routers multicast address, LLMNR and mDNS messages and then sequences of *Neighbor Solicitation/Advertisement* and ping messages) and collects addresses from the responses.

The results shown in the table Tab. 4.4 may look strange at first glance, but

```
[v] Found 5 devices
[i] Device number 1: (Host)
MAC    00:0c:29:2a:8e:72
IPv4   192.168.0.4
IPv6   2001:f:b:a:21d0:d656:c50a:2a8
IPv6   2001:f:b:a:ad75:9d9e:41ba:5299
IPv6   2001:f:b:b::3
IPv6   fe80::e734:2b41:5223:b9c9
[i] Device number 2: (Host)
MAC    00:0c:29:49:04:db
IPv6   2001:f:b:a:2133:775f:f24:9b7d
IPv6   2001:f:b:a:5b68:ae2:e3fc:ee73
IPv6   2001:f:b:b::4
IPv6   fe80::f396:bb03:162c:7a59
[i] Device number 3: (Host)
MAC    00:0c:29:7b:c6:e1
IPv4   192.168.0.5
IPv6   2001:f:b:a:3136:e53d:ee4a:c75a
IPv6   2001:f:b:a:d200:1e15:e46f:1078
IPv6   2001:f:b:b::2
IPv6   fe80::3ea3:c740:13cb:39a3
[i] Device number 4: (Host)
MAC    00:50:56:c0:00:02
IPv6   2001:f:b:a:dd88:d4de:9408:6493
IPv6   fe80::e07:8494:45d9:9d55
IPv6   XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XX1c:c4ee (possible address)
IPv6   XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XX5c:6bbc (possible address)
IPv6   XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XX96:17d3 (possible address)
IPv6   XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXb9:1fd9 (possible address)
IPv6   XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXb9:ad8e (possible address)
[i] Device number 5: (Host)
MAC    ca:01:09:06:00:00
IPv6   2001:f:b:a::1
IPv6   fe80::c801:9ff:fe06:0
[i] Active scan ended
```

Fig. 4.23: Program output of the active mode scenario 4.

they are explained in the section Performance testing script.

Address A	Address B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A
2001:f:b:a:20c:29ff:feb8:a94d	2001:f:b:a:dd88:d4de:9408:6493	1	78 bajty	1	78 bajty	0	0 bajty
2001:f:b:a:20c:29ff:feb8:a94d	ff02::1	3	202 bajty	3	202 bajty	0	0 bajty
2001:f:b:a:2133:775f:f24:9b7d	2001:f:b:a:20c:29ff:feb8:a94d	3	298 bajty	3	298 bajty	0	0 bajty
2001:f:b:a:3136:e53d:ee4a:c75a	ff02::1:ff06:0	12	1,008 KiB	12	1,008 KiB	0	0 bajty
2001:f:b:a::1	2001:f:b:a:20c:29ff:feb8:a94d	3	298 bajty	3	298 bajty	0	0 bajty
2001:f:b:a:ad75:9d9e:41ba:5299	ff02::1:ff06:0	3	258 bajty	3	258 bajty	0	0 bajty
2001:f:b:b::4	ff02::fb	2	342 bajty	2	342 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	2001:f:b:a:dd88:d4de:9408:6493	1	86 bajty	1	86 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	fe80::e07:8494:45d9:9d55	2	164 bajty	2	164 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	ff02::1	7	554 bajty	7	554 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	ff02::1:3	17	1,996 KiB	17	1,996 KiB	0	0 bajty
fe80::20c:29ff:feb8:a94d	ff02::2	2	132 bajty	2	132 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	ff02::fb	15	1,752 KiB	15	1,752 KiB	0	0 bajty
fe80::3ea3:c740:13cb:39a3	fe80::20c:29ff:feb8:a94d	8	1,093 KiB	6	955 bajty	2	164 bajty
fe80::3ea3:c740:13cb:39a3	ff02::1:ff00:2	4	344 bajty	4	344 bajty	0	0 bajty
fe80::3ea3:c740:13cb:39a3	ff02::1:ff06:0	3	258 bajty	3	258 bajty	0	0 bajty
fe80::3ea3:c740:13cb:39a3	ff02::1:ff4a:c75a	4	344 bajty	4	344 bajty	0	0 bajty
fe80::3ea3:c740:13cb:39a3	ff02::1:fff6:1078	4	344 bajty	4	344 bajty	0	0 bajty
fe80::3ea3:c740:13cb:39a3	ff02::1:ffcb:39a3	4	344 bajty	4	344 bajty	0	0 bajty
fe80::3ea3:c740:13cb:39a3	ff02::fb	3	541 bajty	3	541 bajty	0	0 bajty
fe80::c801:9ff:fe06:0	fe80::20c:29ff:feb8:a94d	5	446 bajty	4	360 bajty	1	86 bajty
fe80::c801:9ff:fe06:0	ff02::1	2	236 bajty	2	236 bajty	0	0 bajty
fe80::e734:2b41:5223:b9c9	fe80::20c:29ff:feb8:a94d	8	1,093 KiB	6	955 bajty	2	164 bajty
fe80::e734:2b41:5223:b9c9	ff02::1:ff00:3	4	344 bajty	4	344 bajty	0	0 bajty
fe80::e734:2b41:5223:b9c9	ff02::1:ff0a:2a8	4	344 bajty	4	344 bajty	0	0 bajty
fe80::e734:2b41:5223:b9c9	ff02::1:ff23:b9c9	4	344 bajty	4	344 bajty	0	0 bajty
fe80::e734:2b41:5223:b9c9	ff02::1:ffba:5299	4	344 bajty	4	344 bajty	0	0 bajty
fe80::e734:2b41:5223:b9c9	ff02::fb	3	541 bajty	3	541 bajty	0	0 bajty
fe80::f396:bb03:162c:7a59	2001:f:b:a:20c:29ff:feb8:a94d	2	164 bajty	1	86 bajty	1	78 bajty
fe80::f396:bb03:162c:7a59	fe80::20c:29ff:feb8:a94d	7	626 bajty	5	462 bajty	2	164 bajty
fe80::f396:bb03:162c:7a59	ff02::16	6	1 020 bajty	6	1 020 bajty	0	0 bajty
fe80::f396:bb03:162c:7a59	ff02::1:ff00:4	2	172 bajty	2	172 bajty	0	0 bajty
fe80::f396:bb03:162c:7a59	ff02::1:ff24:9b7d	2	172 bajty	2	172 bajty	0	0 bajty
fe80::f396:bb03:162c:7a59	ff02::1:ff2c:7a59	2	172 bajty	2	172 bajty	0	0 bajty
fe80::f396:bb03:162c:7a59	ff02::1:fffcee73	2	172 bajty	2	172 bajty	0	0 bajty

Fig. 4.24: IPv6 conversations during the active scan in the scenario 4.

Table 4.4: Performance testing of the active mode – 3 runs measured.

Scenario	Average runtime	Average CPU usage	Average RAM usage
Scenario 1	11.288 s	42.835 %	2.532 %
Scenario 2	12.306 s	44.220 %	2.462 %
Scenario 3	11.177 s	35.677 %	2.737 %
Scenario 4	38.266 s	29.537 %	2.794 %
Scenario 5	35.078 s	27.148 %	2.924 %

### 4.4.3 Testing of Aggressive mode of the pnetinspector

In this section, the application will be run in the aggressive mode. Again, as in the case of active mode, all the clients can be already running and have their addresses



```

[i] Device number 1: (Host)
MAC    00:0c:29:2a:8e:72
IPv4   192.168.0.4
IPv6   2001:f:b:a:21d0:d656:c50a:2a8
IPv6   2001:f:b:a:302b:7571:2efa:136a
IPv6   2001:f:b:b::3
IPv6   2001:f:b:b:302b:7571:2efa:136a
IPv6   2001:f:b:b:6923:f0d2:b8a2:8947
IPv6   2001:f:b:c:302b:7571:2efa:136a
IPv6   2001:f:b:c:76c4:5af7:93f0:edab
IPv6   2001:f:b:d:99d:73ee:6586:9433
IPv6   2001:f:b:d:302b:7571:2efa:136a
IPv6   2001:f:b:e:302b:7571:2efa:136a
IPv6   2001:f:b:e:3d61:bc08:df46:914f
IPv6   fe80::e734:2b41:5223:b9c9

```

Fig. 4.25: Program output of the active mode scenario 5 (Windows 11).

assigned (generated). Kali machine will be acting as a fake router in this mode with certain prefix. The chosen prefix is *2001:f:b:f::/64*.

The scenarios are the same as in the testing of active mode.

1. **The application will be started when there is no other IPv6 device on the network.**

There is no other IPv6 device, therefore Kali captures no response. It starts with scanning the network in the active mode and sends the same packet sequence as discussed in the Active mode testing section. After initial scanning, Kali takes over the role of a fake router. It starts distributing the *Router Advertisement* messages with high priority and the prefix set. Then it sends *Neighbor Advertisement* packets, where it proclaims itself as a router by setting particular flag. It also configures the *Solicited* and *Override* flags to 1. The MLDv1 queries and MLDv2 reports as well as the ICMPv6 echo messages are transmitted. The conversations are shown in the Fig. 4.26 and the runtime results in the Tab. 4.5.

2. **The application will be started when other devices are on (except for the router) and have only Link-Local address.**

The program starts with the active scan, therefore MLD queries and reports are exchanged and the 3 machines are encountered with their respective Link-Local addresses. After number of ICMPv6 (*Neighbor Solicitation/Advertisement*) messages, LLMNR and mDNS messages, Kali takes the role of a router

Address A	Address B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A
fe80::20c:29ff:feb8:a94d	2001:fb:fe07:8494:45d9:9d55	3	202 bajty	3	202 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	fe80::20c:29ff:feb8:a94d	2	172 bajty	2	172 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	fe80::e07:8494:45d9:9d55	11	946 bajty	11	946 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	ff02::1	26	2,379 KiB	26	2,379 KiB	0	0 bajty
fe80::20c:29ff:feb8:a94d	ff02::16	5	590 bajty	5	590 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	ff02::13	4	608 bajty	4	608 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	ff02::2	4	264 bajty	4	264 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	ff02::fb	4	608 bajty	4	608 bajty	0	0 bajty

Fig. 4.26: IPv6 conversations during the aggressive scan in the scenario 1.

and announces this in the *Neighbor Advertisement* message using the corresponding flags. It also includes the prefix to the *Router Advertisement* messages. After series of MLD messages, Windows 10 and Windows 11 machines announce their IPv6 generated address from the prefix sent using the *Neighbor Advertisement* messages for both permanent and temporary addresses. The Ubuntu addresses are obtained from the mDNS messages. The output of the program is shown in the Fig. 4.27 and the conversations are shown in the Fig. 4.28.

```
[v] Found 4 devices
[i] Device number 1: (Host)
    MAC    00:0c:29:2a:8e:72
    IPv4   169.254.99.223
    IPv6   2001:f:b:f:523d:a60:7859:766a
    IPv6   2001:f:b:f:74ef:bcaf:a80a:bfbc
    IPv6   fe80::e734:2b41:5223:b9c9
[i] Device number 2: (Host)
    MAC    00:0c:29:49:04:db
    IPv6   2001:f:b:f:bea0:e26e:f58c:7035
    IPv6   2001:f:b:f:e3e8:b4d2:26b5:634d
    IPv6   fe80::f396:bb03:162c:7a59
[i] Device number 3: (Host)
    MAC    00:0c:29:7b:c6:e1
    IPv4   169.254.198.169
    IPv6   2001:f:b:f:b9a:ed2d:fb19:7d9b
    IPv6   2001:f:b:f:a4fe:ffa2:a251:6c02
    IPv6   fe80::3ea3:c740:13cb:39a3
[i] Device number 4: (Host)
    MAC    00:50:56:c0:00:02
    IPv4   169.254.52.37
    IPv6   2001:f:b:f:8d18:776d:29d6:10ea
    IPv6   fe80::e07:8494:45d9:9d55
    IPv6   XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XX34:71e7 (possible address)
```

Fig. 4.27: Program output of the aggressive mode scenario 2.



Address A	Address B	Packets	Bytes	Total Packets	Percent Filtered	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A
2001:fb:f523d:a60:7859:766a	ff02::1	1	86 bajty	1	100.00%	1	86 bajty	0	0 bajty
2001:fb:f74ef:bcafa80a:bfbcb	ff02::1	1	86 bajty	1	100.00%	1	86 bajty	0	0 bajty
2001:fb:fa4fe:ffa2:a251:6c02	ff02::1	1	86 bajty	1	100.00%	1	86 bajty	0	0 bajty
2001:fb:fb9aed2d:fb19:7d9b	ff02::1	1	86 bajty	1	100.00%	1	86 bajty	0	0 bajty
2001:fb:fe3e8:b4d2:26b5:634d	ff02::fb	19	3,813 KiB	19	100.00%	19	3,813 KiB	0	0 bajty
fe80::20c:29ff:feb8:a94d	2001:fb:fb3ea3:c740:13cb:39a3	3	202 bajty	3	100.00%	3	202 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	2001:fb:fe07:8494:45d9:9d55	3	202 bajty	3	100.00%	3	202 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	2001:fb:fe734:2b41:5223:b9c9	3	202 bajty	3	100.00%	3	202 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	2001:fb:ff396:bb03:162c:7a59	3	202 bajty	3	100.00%	3	202 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	fe80::20c:29ff:feb8:a94d	6	516 bajty	6	100.00%	6	516 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	fe80::3ea3:c740:13cb:39a3	14	1,176 KiB	14	100.00%	11	946 bajty	3	258 bajty
fe80::20c:29ff:feb8:a94d	fe80::e07:8494:45d9:9d55	11	938 bajty	11	100.00%	11	938 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	fe80::e734:2b41:5223:b9c9	11	946 bajty	11	100.00%	8	688 bajty	3	258 bajty
fe80::20c:29ff:feb8:a94d	fe80::f396:bb03:162c:7a59	13	1,146 KiB	13	100.00%	4	344 bajty	9	830 bajty
fe80::20c:29ff:feb8:a94d	ff02::1	27	2,486 KiB	27	100.00%	27	2,486 KiB	0	0 bajty
fe80::20c:29ff:feb8:a94d	ff02::16	4	520 bajty	4	100.00%	4	520 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	ff02::1:3	24	3,176 KiB	24	100.00%	24	3,176 KiB	0	0 bajty
fe80::20c:29ff:feb8:a94d	ff02::2	4	264 bajty	4	100.00%	4	264 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	ff02::fb	24	3,211 KiB	24	100.00%	24	3,211 KiB	0	0 bajty
fe80::3ea3:c740:13cb:39a3	ff02::16	2	300 bajty	2	100.00%	2	300 bajty	0	0 bajty
fe80::3ea3:c740:13cb:39a3	ff02::1:3	6	534 bajty	6	100.00%	6	534 bajty	0	0 bajty
fe80::3ea3:c740:13cb:39a3	ff02::1:ff19:7d9b	4	344 bajty	4	100.00%	4	344 bajty	0	0 bajty
fe80::3ea3:c740:13cb:39a3	ff02::1:ff51:6c02	4	344 bajty	4	100.00%	4	344 bajty	0	0 bajty
fe80::3ea3:c740:13cb:39a3	ff02::1:ffb8:a94d	4	344 bajty	4	100.00%	4	344 bajty	0	0 bajty
fe80::3ea3:c740:13cb:39a3	ff02::1:ffcb:39a3	6	516 bajty	6	100.00%	6	516 bajty	0	0 bajty
fe80::3ea3:c740:13cb:39a3	ff02::2	1	86 bajty	1	100.00%	1	86 bajty	0	0 bajty
fe80::3ea3:c740:13cb:39a3	ff02::c	1	86 bajty	1	100.00%	1	86 bajty	0	0 bajty
fe80::3ea3:c740:13cb:39a3	ff02::fb	8	880 bajty	8	100.00%	8	880 bajty	0	0 bajty
fe80::e734:2b41:5223:b9c9	ff02::16	2	300 bajty	2	100.00%	2	300 bajty	0	0 bajty
fe80::e734:2b41:5223:b9c9	ff02::1:3	5	448 bajty	5	100.00%	5	448 bajty	0	0 bajty
fe80::e734:2b41:5223:b9c9	ff02::1:ff0a:bfbcb	6	516 bajty	6	100.00%	6	516 bajty	0	0 bajty
fe80::e734:2b41:5223:b9c9	ff02::1:ff23:b9c9	6	516 bajty	6	100.00%	6	516 bajty	0	0 bajty
fe80::e734:2b41:5223:b9c9	ff02::1:ff59:766a	6	516 bajty	6	100.00%	6	516 bajty	0	0 bajty
fe80::e734:2b41:5223:b9c9	ff02::1:ffb8:a94d	3	258 bajty	3	100.00%	3	258 bajty	0	0 bajty
fe80::e734:2b41:5223:b9c9	ff02::fb	7	794 bajty	7	100.00%	7	794 bajty	0	0 bajty
fe80::f396:bb03:162c:7a59	ff02::16	30	4,277 KiB	30	100.00%	30	4,277 KiB	0	0 bajty
fe80::f396:bb03:162c:7a59	ff02::1:ff2c:7a59	4	344 bajty	4	100.00%	4	344 bajty	0	0 bajty
fe80::f396:bb03:162c:7a59	ff02::1:ff8c:7035	2	172 bajty	2	100.00%	2	172 bajty	0	0 bajty
fe80::f396:bb03:162c:7a59	ff02::1:ffb5:634d	2	172 bajty	2	100.00%	2	172 bajty	0	0 bajty
fe80::f396:bb03:162c:7a59	ff02::1:ffb8:a94d	1	86 bajty	1	100.00%	1	86 bajty	0	0 bajty
fe80::f396:bb03:162c:7a59	ff02::2	1	62 bajty	1	100.00%	1	62 bajty	0	0 bajty
fe80::f396:bb03:162c:7a59	ff02::fb	6	672 bajty	6	100.00%	6	672 bajty	0	0 bajty

Fig. 4.28: IPv6 conversations during the aggressive scan in the scenario 2.

3. **The application will be started when other devices are on (except for the router) and have the Link-Local and static global addresses.** After MLD messages during the active scan, the program has information about the Link-Local addresses and possible (static) addresses. Message sequence is very similar to the one during the scenario 2. Static IPv6 address of the Ubuntu machine is discovered in the mDNS messages. In this scenario, both Windows machines did not use the *Neighbor Advertisement* messages to report their static IPv6 addresses. The static IPv6 addresses and the ones generated from the prefix sent from Kali were all included inside the mDNS messages, but the program was not able to recognize them. From the point of view of configuration, the virtual machines set their default gateway to the address of Kali only temporarily. After Kali stops sending the messages, it is removed. The output of program is shown in the Fig. 4.29 and the conversations are shown in the Fig. 4.30.

```
[v] Found 4 devices
[i] Device number 1: (Host)
    MAC    00:0c:29:2a:8e:72
    IPv4   169.254.99.223
    IPv6   2001:f:b:f:523d:a60:7859:766a
    IPv6   2001:f:b:f:68ca:231e:fc80:2d65
    IPv6   fe80::e734:2b41:5223:b9c9
    IPv6   XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XX00:0003 (possible address)
[i] Device number 2: (Host)
    MAC    00:0c:29:49:04:db
    IPv6   2001:f:b:b::4
    IPv6   2001:f:b:f:e3e8:b4d2:26b5:634d
    IPv6   2001:f:b:f:f603:44ba:8849:bef
    IPv6   fe80::f396:bb03:162c:7a59
[i] Device number 3: (Host)
    MAC    00:0c:29:7b:c6:e1
    IPv4   169.254.198.169
    IPv6   2001:f:b:f:b9a:ed2d:fb19:7d9b
    IPv6   2001:f:b:f:ed84:3ed6:71ae:da7a
    IPv6   fe80::3ea3:c740:13cb:39a3
    IPv6   XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XX00:0002 (possible address)
[i] Device number 4: (Host)
    MAC    00:50:56:c0:00:02
    IPv4   169.254.52.37
    IPv6   2001:f:b:f:8d18:776d:29d6:10ea
    IPv6   fe80::e07:8494:45d9:9d55
    IPv6   XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XX34:71e7 (possible address)
```

Fig. 4.29: Program output of the aggressive mode scenario 3.

Address A	Address B	Packets	Bytes	Total Packets	Percent Filtered	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A
2001:fb:bb:4	ff02:fb	22	3,923 KiB	22	100.00%	22	3,923 KiB	0	0 bajty
2001:fb:bf:523d:a60:7859:766a	ff02::1	1	86 bajty	1	100.00%	1	86 bajty	0	0 bajty
2001:fb:bf:68ca:231e:fc80:2d65	ff02::1	1	86 bajty	1	100.00%	1	86 bajty	0	0 bajty
2001:fb:bf:b9aed2:ffb19:7d9b	ff02::1	1	86 bajty	1	100.00%	1	86 bajty	0	0 bajty
2001:fb:fed84:3ed6:71aeda7a	ff02::1	1	86 bajty	1	100.00%	1	86 bajty	0	0 bajty
fe80:20c29ff:feb8:a94d	2001:fb:bf:3ea3:c740:13cb:39a3	3	202 bajty	3	100.00%	3	202 bajty	0	0 bajty
fe80:20c29ff:feb8:a94d	2001:fb:fe07:8494:45d9:9d55	3	202 bajty	3	100.00%	3	202 bajty	0	0 bajty
fe80:20c29ff:feb8:a94d	2001:fb:fe734:2b41:5223:b9c9	3	202 bajty	3	100.00%	3	202 bajty	0	0 bajty
fe80:20c29ff:feb8:a94d	2001:fb:ff396:bb03:162c:7a59	3	202 bajty	3	100.00%	3	202 bajty	0	0 bajty
fe80:20c29ff:feb8:a94d	fe80:20c29ff:feb8:a94d	6	516 bajty	6	100.00%	6	516 bajty	0	0 bajty
fe80:20c29ff:feb8:a94d	fe80:3ea3:c740:13cb:39a3	14	1,176 KiB	14	100.00%	11	946 bajty	3	258 bajty
fe80:20c29ff:feb8:a94d	fe80:e07:8494:45d9:9d55	11	938 bajty	11	100.00%	11	938 bajty	0	0 bajty
fe80:20c29ff:feb8:a94d	fe80:e734:2b41:5223:b9c9	12	1,008 KiB	12	100.00%	9	774 bajty	3	258 bajty
fe80:20c29ff:feb8:a94d	fe80:f396:bb03:162c:7a59	14	1,230 KiB	14	100.00%	5	422 bajty	9	838 bajty
fe80:20c29ff:feb8:a94d	ff02:1	27	2,486 KiB	27	100.00%	27	2,486 KiB	0	0 bajty
fe80:20c29ff:feb8:a94d	ff02:16	4	520 bajty	4	100.00%	4	520 bajty	0	0 bajty
fe80:20c29ff:feb8:a94d	ff02::13	21	2,924 KiB	21	100.00%	21	2,924 KiB	0	0 bajty
fe80:20c29ff:feb8:a94d	ff02:2	4	264 bajty	4	100.00%	4	264 bajty	0	0 bajty
fe80:20c29ff:feb8:a94d	ff02:fb	21	2,941 KiB	21	100.00%	21	2,941 KiB	0	0 bajty
fe80:3ea3:c740:13cb:39a3	ff02:16	2	340 bajty	2	100.00%	2	340 bajty	0	0 bajty
fe80:3ea3:c740:13cb:39a3	ff02:1:2	5	785 bajty	5	100.00%	5	785 bajty	0	0 bajty
fe80:3ea3:c740:13cb:39a3	ff02:1:3	4	362 bajty	4	100.00%	4	362 bajty	0	0 bajty
fe80:3ea3:c740:13cb:39a3	ff02:1:ff00:2	5	430 bajty	5	100.00%	5	430 bajty	0	0 bajty
fe80:3ea3:c740:13cb:39a3	ff02:1:ff19:7d9b	5	430 bajty	5	100.00%	5	430 bajty	0	0 bajty
fe80:3ea3:c740:13cb:39a3	ff02:1:ffaeda7a	5	430 bajty	5	100.00%	5	430 bajty	0	0 bajty
fe80:3ea3:c740:13cb:39a3	ff02:1:ffb8:a94d	4	344 bajty	4	100.00%	4	344 bajty	0	0 bajty
fe80:3ea3:c740:13cb:39a3	ff02:1:ffc3:39a3	5	430 bajty	5	100.00%	5	430 bajty	0	0 bajty
fe80:3ea3:c740:13cb:39a3	ff02:fb	6	764 bajty	6	100.00%	6	764 bajty	0	0 bajty
fe80:e734:2b41:5223:b9c9	ff02:16	2	340 bajty	2	100.00%	2	340 bajty	0	0 bajty
fe80:e734:2b41:5223:b9c9	ff02:1:3	5	448 bajty	5	100.00%	5	448 bajty	0	0 bajty
fe80:e734:2b41:5223:b9c9	ff02:1:ff00:3	6	516 bajty	6	100.00%	6	516 bajty	0	0 bajty
fe80:e734:2b41:5223:b9c9	ff02:1:ff23:b9c9	6	516 bajty	6	100.00%	6	516 bajty	0	0 bajty
fe80:e734:2b41:5223:b9c9	ff02:1:ff59:766a	4	344 bajty	4	100.00%	4	344 bajty	0	0 bajty
fe80:e734:2b41:5223:b9c9	ff02:1:ff80:2d65	4	344 bajty	4	100.00%	4	344 bajty	0	0 bajty
fe80:e734:2b41:5223:b9c9	ff02:1:ffb8:a94d	3	258 bajty	3	100.00%	3	258 bajty	0	0 bajty
fe80:e734:2b41:5223:b9c9	ff02:fb	7	850 bajty	7	100.00%	7	850 bajty	0	0 bajty
fe80:f396:bb03:162c:7a59	ff02:16	27	4,424 KiB	27	100.00%	27	4,424 KiB	0	0 bajty
fe80:f396:bb03:162c:7a59	ff02:1:ff00:4	4	344 bajty	4	100.00%	4	344 bajty	0	0 bajty
fe80:f396:bb03:162c:7a59	ff02:1:ff2c:7a59	4	344 bajty	4	100.00%	4	344 bajty	0	0 bajty
fe80:f396:bb03:162c:7a59	ff02:1:ff49:bef	2	172 bajty	2	100.00%	2	172 bajty	0	0 bajty
fe80:f396:bb03:162c:7a59	ff02:1:ffb5:634d	2	172 bajty	2	100.00%	2	172 bajty	0	0 bajty
fe80:f396:bb03:162c:7a59	ff02:1:ffb8:a94d	1	86 bajty	1	100.00%	1	86 bajty	0	0 bajty

Fig. 4.30: IPv6 conversations during aggressive scan in the scenario 3.

- The application will be started when all the devices are powered on (containing all the previous addresses) and the router propagates 1 prefix.

The conversation starts with MLD (both version 1 and version 2) messages exchange. Then the Kali machine has opportunity to gather all the addresses from the mDNS response messages. The conversations consisting of *Neighbor Solicitation/Advertisement* messages are also observable. But until this moment, this is the active mode check, therefore Kali does not set flags inside the *Neighbor Advertisement* messages that would indicate the router. When the aggressive mode starts, Kali distributes the *Router Advertisement* messages with particular prefix. The router still propagates its *Router Advertisement* messages, but with the default preference (*medium*). Kali sets the preference to *high*, therefore it should be selected as the default router by the clients. Kali also informs specifically the router using *Neighbor Advertisement* message that he is the router as well. Both Windows machines have set both addresses (of

the Kali and the Cisco router) as their default gateway. The Ubuntu stores only one address for its profile and there can be seen changes over time. The address gets overwritten as the *Advertisement* messages come, because Kali has higher preference. Router sends the *Router Advertisement* messages every 10 seconds and Kali sends the last *Router Advertisement* message with the lifetime 0, therefore the final address is always the one of the router. After the end of the script, both Windows machines store only the default gateway of the router. The generated IPv6 addresses are preserved. The output of the program is shown in the Fig. 4.31 and the conversation samples are shown in the Fig. 4.32.

**5. The application will be started when all the devices are powered on (containing all the previous addresses) and the router propagates 5 prefixes.**

As in other scenarios, the active mode begins with MLD queries and reports. The program obtains the basic information about the addresses that the devices are listening to (part of multicast group) and are configured with. Then the ping is issued from the program to the *all-node multicast* address (*ff02::1*). Only the Ubuntu machine answers to this from most of its generated global addresses (both permanent and temporary). Then the series of transmissions of *Neighbor Solicitation/Advertisement* messages to obtain the MAC addresses of the devices on the same link follows. Both *Solicitation* and *Advertisement* messages are generated for individual global addresses. Then Kali starts to act as a fake router. A huge number of LLMNR and mDNS messages is exchanged between Kali and other clients, where the clients include their addresses in the payload of the answer. During the packet analysis, it was observed that Kali sent the *Neighbor Advertisement* messages to himself. Kali also sent messages to the *VMnet* interface over which it is connected with the physical (host) machine. The default gateway setting on the virtual machines works in the same way is in scenario 4, i.e. Windows clients set 2 default routers (Cisco and Kali) as long as Kali sends messages about its router role, then only Cisco remains. Ubuntu holds only 1 default router in its profile, therefore as long as Kali acts as the fake router, its address is used (high priority) and after the end of the program run is replaced with Cisco. The output of the program (client addresses) is shown in the Fig. 4.33. The router output is shown in the Fig. 4.34.

The results of the aggressive mode are shown in the Tab. 4.5. It can be seen that with higher number of inputs for the program (increasing number of addresses), the average runtime grows larger. The same can be said about the CPU and RAM usage. The fact that the average CPU usage in the 5th scenario is similar to the

```
[v] Found 5 devices
[i] Device number 1: (Host)
MAC    00:0c:29:2a:8e:72
IPv4   192.168.0.3
IPv6   2001:f:b:a:21d0:d656:c50a:2a8
IPv6   2001:f:b:a:35c0:cd0e:42e3:d909
IPv6   2001:f:b:b::3
IPv6   2001:f:b:f:35c0:cd0e:42e3:d909
IPv6   2001:f:b:f:523d:a60:7859:766a
IPv6   fe80::e734:2b41:5223:b9c9
[i] Device number 2: (Host)
MAC    00:0c:29:49:04:db
IPv4   192.168.0.4
IPv6   2001:f:b:a:1176:f4:21bd:678c
IPv6   2001:f:b:a:5b68:ae2:e3fc:ee73
IPv6   2001:f:b:b::4
IPv6   2001:f:b:f:979f:4f04:a5f9:aeae
IPv6   2001:f:b:f:e3e8:b4d2:26b5:634d
IPv6   fe80::f396:bb03:162c:7a59
[i] Device number 3: (Host)
MAC    00:0c:29:7b:c6:e1
IPv4   192.168.0.5
IPv6   2001:f:b:a:a0d8:fe4f:2d13:1804
IPv6   2001:f:b:a:d200:1e15:e46f:1078
IPv6   2001:f:b:b::2
IPv6   2001:f:b:f:b9a:ed2d:fb19:7d9b
IPv6   2001:f:b:f:a0d8:fe4f:2d13:1804
IPv6   fe80::3ea3:c740:13cb:39a3
[i] Device number 4: (Host)
MAC    00:50:56:c0:00:02
IPv4   169.254.52.37
IPv6   2001:f:b:a:150b:9095:b080:74fe
IPv6   fe80::e07:8494:45d9:9d55
IPv6   XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XX34:71e7 (possible address)
IPv6   XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXb9:1fd9 (possible address)
[i] Device number 5: (Preferred router)
MAC    ca:01:09:06:00:00
IPv6   2001:f:b:a::1
IPv6   fe80::c801:9ff:fe06:0
```

Fig. 4.31: Program output of the aggressive mode scenario 4.

previous scenarios, even though the runtime is at least two times larger, gives nice picture about the overall performance (higher load is distributed over larger time interval).

Address A	Address B	Packets	Bytes	Total Packets	Percent Filtered	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A
2001:fb:a:1176:f4:21bd:678c	2001:fb:a:20c:29ff:feb8:a94d	6	596 bajty	6	100.00%	6	596 bajty	0	0 bajty
2001:fb:a:20c:29ff:feb8:a94d	2001:fb:a:150b:9095:b080:74fe	1	78 bajty	1	100.00%	1	78 bajty	0	0 bajty
2001:fb:a:20c:29ff:feb8:a94d	2001:fb:a:3ea3:c740:13cb:39a3	3	202 bajty	3	100.00%	3	202 bajty	0	0 bajty
2001:fb:a:20c:29ff:feb8:a94d	2001:fb:a:1	7	730 bajty	7	100.00%	1	86 bajty	6	644 bajty
2001:fb:a:20c:29ff:feb8:a94d	2001:fb:a:c801:9ff:fe06:0	3	202 bajty	3	100.00%	3	202 bajty	0	0 bajty
2001:fb:a:20c:29ff:feb8:a94d	2001:fb:a:e07:8494:45d9:9d55	3	202 bajty	3	100.00%	3	202 bajty	0	0 bajty
2001:fb:a:20c:29ff:feb8:a94d	2001:fb:a:e734:2b41:5223:b9c9	3	202 bajty	3	100.00%	3	202 bajty	0	0 bajty
2001:fb:a:20c:29ff:feb8:a94d	2001:fb:a:f396:bb03:162c:7a59	3	202 bajty	3	100.00%	3	202 bajty	0	0 bajty
2001:fb:a:20c:29ff:feb8:a94d	2001:fb:f3ea3:c740:13cb:39a3	3	202 bajty	3	100.00%	3	202 bajty	0	0 bajty
2001:fb:a:20c:29ff:feb8:a94d	2001:fb:fc801:9ff:fe06:0	3	202 bajty	3	100.00%	3	202 bajty	0	0 bajty
2001:fb:a:20c:29ff:feb8:a94d	2001:fb:fe07:8494:45d9:9d55	3	202 bajty	3	100.00%	3	202 bajty	0	0 bajty
2001:fb:a:20c:29ff:feb8:a94d	2001:fb:fe734:2b41:5223:b9c9	3	202 bajty	3	100.00%	3	202 bajty	0	0 bajty
2001:fb:a:20c:29ff:feb8:a94d	2001:fb:ff396:bb03:162c:7a59	3	202 bajty	3	100.00%	3	202 bajty	0	0 bajty
2001:fb:a:20c:29ff:feb8:a94d	ff02:1	6	404 bajty	6	100.00%	6	404 bajty	0	0 bajty
2001:fb:a:1	ff02:1:ff13:1804	1	86 bajty	1	100.00%	1	86 bajty	0	0 bajty
2001:fb:a:1	ff02:1:ffb8:a94d	1	86 bajty	1	100.00%	1	86 bajty	0	0 bajty
2001:fb:b:4	ff02:fb	18	4,065 KIB	18	100.00%	18	4,065 KIB	0	0 bajty
2001:fb:f35c:cd0e:42e3:d909	2001:2030:21:3e73:fc52	5	430 bajty	5	100.00%	5	430 bajty	0	0 bajty
2001:fb:f35c:cd0e:42e3:d909	2001:2030:21:3e73:fc81	8	688 bajty	8	100.00%	8	688 bajty	0	0 bajty
2001:fb:f979:f4f04:a5f9:aeae	2620:2d:4000:1:23	5	470 bajty	5	100.00%	5	470 bajty	0	0 bajty
2001:fb:fa0d8:fe4f:2d13:1804	2001:2030:21:3e73:fc52	5	430 bajty	5	100.00%	5	430 bajty	0	0 bajty
2001:fb:fa0d8:fe4f:2d13:1804	2001:2030:21:3e73:fc81	5	430 bajty	5	100.00%	5	430 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	2001:fb:a:150b:9095:b080:74fe	1	86 bajty	1	100.00%	1	86 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	2001:fb:a:3ea3:c740:13cb:39a3	3	202 bajty	3	100.00%	3	202 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	2001:fb:a:1	4	344 bajty	4	100.00%	4	344 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	2001:fb:a:c801:9ff:fe06:0	3	202 bajty	3	100.00%	3	202 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	2001:fb:a:e07:8494:45d9:9d55	3	202 bajty	3	100.00%	3	202 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	2001:fb:a:e734:2b41:5223:b9c9	3	202 bajty	3	100.00%	3	202 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	2001:fb:a:f396:bb03:162c:7a59	3	202 bajty	3	100.00%	3	202 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	2001:fb:f3ea3:c740:13cb:39a3	3	202 bajty	3	100.00%	3	202 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	2001:fb:fc801:9ff:fe06:0	3	202 bajty	3	100.00%	3	202 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	2001:fb:fe07:8494:45d9:9d55	3	202 bajty	3	100.00%	3	202 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	2001:fb:fe734:2b41:5223:b9c9	3	202 bajty	3	100.00%	3	202 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	2001:fb:ff396:bb03:162c:7a59	3	202 bajty	3	100.00%	3	202 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	fe80::20c:29ff:feb8:a94d	11	946 bajty	11	100.00%	11	946 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	fe80::e07:8494:45d9:9d55	18	1,504 KIB	18	100.00%	18	1,504 KIB	0	0 bajty
fe80::20c:29ff:feb8:a94d	ff02:1	27	2,486 KIB	27	100.00%	27	2,486 KIB	0	0 bajty
fe80::20c:29ff:feb8:a94d	ff02:16	4	520 bajty	4	100.00%	4	520 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	ff02:1:3	83	9,543 KIB	83	100.00%	83	9,543 KIB	0	0 bajty
fe80::20c:29ff:feb8:a94d	ff02:2	4	264 bajty	4	100.00%	4	264 bajty	0	0 bajty
fe80::20c:29ff:feb8:a94d	ff02:fb	71	8,043 KIB	71	100.00%	71	8,043 KIB	0	0 bajty
fe80::3ea3:c740:13cb:39a3	fe80::20c:29ff:feb8:a94d	42	6,357 KIB	42	100.00%	28	5,197 KIB	14	1,160 KIB

Fig. 4.32: IPv6 conversations during the aggressive scan in the scenario 4.

Table 4.5: Performance testing of the aggressive mode – 3 runs measured.

Scenario	Average runtime	Average CPU usage	Average RAM usage
Scenario 1	31.750 s	17.804 %	2.341 %
Scenario 2	32.802 s	14.301 %	2.465 %
Scenario 3	32.967 s	16.532 %	2.478 %
Scenario 4	62.822 s	15.272 %	2.622 %
Scenario 5	135.658 s	16.763 %	2.789 %

## 4.5 ptnetinspector adjustments

In this section, discovered shortcomings and proposed changes for the *ptnetinspector* tool are discussed.



```
[i] Device number 1: (Host)
MAC 00:0c:29:2a:8e:72
IPv4 192.168.0.4
IPv6 2001:f:b:a:21d0:d656:c50a:2a8
IPv6 2001:f:b:a:2d61:fd7:2af0:744f
IPv6 2001:f:b:b::3
IPv6 2001:f:b:b:2d61:fd7:2af0:744f
IPv6 2001:f:b:b:6923:f0d2:b8a2:8947
IPv6 2001:f:b:c:2d61:fd7:2af0:744f
IPv6 2001:f:b:c:76c4:5af7:93f0:edab
IPv6 2001:f:b:d:99d:73ee:6586:9433
IPv6 2001:f:b:d:2d61:fd7:2af0:744f
IPv6 2001:f:b:e:2d61:fd7:2af0:744f
IPv6 2001:f:b:e:3d61:bc08:df46:914f
IPv6 2001:f:b:f:523d:a60:7859:766a
IPv6 2001:f:b:f:a4c4:21a6:87ef:36a
IPv6 fe80::e734:2b41:5223:b9c9

[i] Device number 2: (Host)
MAC 00:0c:29:49:04:db
IPv6 2001:f:b:a:12c9:3698:7453:3e40
IPv6 2001:f:b:a:5b68:ae2:e3fc:ee73
IPv6 2001:f:b:b::4
IPv6 2001:f:b:b:2c9f:30f0:36c9:4dec
IPv6 2001:f:b:b:f88c:b8c1:2156:bffb
IPv6 2001:f:b:c:15d9:a9ea:be32:5fb5
IPv6 2001:f:b:c:94e3:d352:bf6c:126d
IPv6 2001:f:b:d:74e6:fe6c:adb1:5417
IPv6 2001:f:b:d:7532:1027:f84a:8955
IPv6 2001:f:b:e:806d:dc6a:9f4a:5923
IPv6 2001:f:b:e:e63b:2d66:c1b4:94ab
IPv6 2001:f:b:f:307f:43af:ed62:851c
IPv6 2001:f:b:f:e3e8:b4d2:26b5:634d
IPv6 fe80::f396:bb03:162c:7a59

[i] Device number 3: (Host)
MAC 00:0c:29:7b:c6:e1
IPv4 192.168.0.3
IPv6 2001:f:b:a:d200:1e15:e46f:1078
IPv6 2001:f:b:a:d979:4317:8f19:173f
IPv6 2001:f:b:b::2
IPv6 2001:f:b:b:700e:e565:47a9:2cc6
IPv6 2001:f:b:b:d979:4317:8f19:173f
IPv6 2001:f:b:c:28ed:ad9a:593b:d729
IPv6 2001:f:b:c:d979:4317:8f19:173f
IPv6 2001:f:b:d:a0c6:7ca7:e983:c9e9
IPv6 2001:f:b:d:d979:4317:8f19:173f
IPv6 2001:f:b:e:c8c:8ea1:ffaa:673d
IPv6 2001:f:b:e:d979:4317:8f19:173f
IPv6 2001:f:b:f:b9a:ed2d:fb19:7d9b
IPv6 2001:f:b:f:d979:4317:8f19:173f
IPv6 fe80::3ea3:c740:13cb:39a3
```

Fig. 4.33: Program output of the aggressive mode scenario 5 (clients).

### 4.5.1 mDNS payload reading

After constructing the *verifyAddresses* script, it was found that the *ptnetinspector* does not analyze payload of the mDNS messages, where information about the addresses of device are carried. This fault was registered only in the passive mode. Active and aggressive modes were running correctly. After reporting the problem, it was fixed with version 17 of the *ptnetinspector*.

```
[i] Device number 5: (Preferred router)
MAC    ca:01:09:06:00:00
IPv6   2001:f:b:a::1
IPv6   2001:f:b:b::1
IPv6   2001:f:b:c::1
IPv6   2001:f:b:d::1
IPv6   2001:f:b:e::1
IPv6   fe80::c801:9ff:fe06:0
```

Fig. 4.34: Program output of the aggressive mode scenario 5 (router).

The result of omitting the addresses is displayed in the Fig. 4.35. The synchronization of the two scripts was not established perfectly at this moment (described in the Timestamp addition section), which is the reason why so many nodes and addresses were not found by the *ptnetinspector*. Nevertheless, the mDNS addresses were still supposed to be processed as they appeared during the passive mode listening of the *ptnetinspector*. The Wireshark screenshot is shown in the Fig. 4.36 to display where the addresses appeared. It can be clearly seen that both addresses are placed inside the mDNS payload of the response message. The third address was captured by the *ptnetinspector* in a form of the source address in other packets.

```
Device: f8:ff:c2:3d:c2:ef
IP addresses: {'100.64.133.143'}
Device: fa:eb:1a:f5:b2:e7
IP addresses: {'100.64.134.89', 'fe80::143b:260d:29de:e4c'}
Device: fe:be:0d:b9:2a:69
IP addresses: {'100.64.134.149', 'fe80::1c12:ac0b:48e1:4113'}
Device: fe:f0:65:f5:d0:dd
IP addresses: {'100.64.131.212'}
#####

#####
fe:f9:03:c3:7a:10 was not captured by the ptNetInspector.
The address 2001:67c:1220:98a1:f124:24da:4d01:1914 was not captured by the ptNetInspector for the 8c:b8:7e:ff:b8:74 node.
The address 2001:67c:1220:98a1:e34c:cd82:57d:78fb was not captured by the ptNetInspector for the 8c:b8:7e:ff:b8:74 node.
38:fc:98:63:8a:b7 was not captured by the ptNetInspector.
02:d8:ef:95:57:6f was not captured by the ptNetInspector.
ba:f5:12:1c:53:a6 was not captured by the ptNetInspector.
14:85:7f:14:e2:5e was not captured by the ptNetInspector.
The address fe80::698:7cd1:f289:78ae was not captured by the ptNetInspector for the f8:5e:a0:55:c3:91 node.
ac:d5:64:48:29:d9 was not captured by the ptNetInspector.
e0:aa:96:7f:69:6c was not captured by the ptNetInspector.
The address 100.64.135.52 was not captured by the ptNetInspector for the e8:84:a5:6c:3d:97 node.
The address fe80::1017:8c43:f480:d831 was not captured by the ptNetInspector for the 4c:1d:96:39:de:83 node.
60:f6:77:96:c7:47 was not captured by the ptNetInspector.
The address fe80::639e:13eb:2227:27a4 was not captured by the ptNetInspector for the 3c:55:76:e9:01:2f node.
f4:c8:8a:76:99:fe was not captured by the ptNetInspector.
The address 100.64.129.138 was not captured by the ptNetInspector for the de:9d:7f:89:81:7c node.
```

Fig. 4.35: mDNS APDU (Application Protocol Data Unit) not processed in the passive mode by the *ptnetinspector*.



```

# ipv6.addr== fe80::c0ad:ac41:b2:1db9 and mdns
No.    Time           Source           Destination      Protocol Length Info
-----
459 1.269150      fe80::c0ad:ac41:b2:1db9 ff02::fb        MDNS          111 Standard query response 0x0000 A 100.64.129.234
460 1.269150      fe80::c0ad:ac41:b2:1db9 ff02::fb        MDNS          179 Standard query response 0x0000 AAAA 2001:67c:1220:98a1:e34c:cd82:57d:78fb AAAA 2001:67c:1220:98a1:f124:24da:4d01:1914
498 1.243573      fe80::c0ad:ac41:b2:1db9 ff02::fb        MDNS          111 Standard query response 0x0000 A 100.64.129.234
498 1.243575      fe80::c0ad:ac41:b2:1db9 ff02::fb        MDNS          179 Standard query response 0x0000 AAAA 2001:67c:1220:98a1:e34c:cd82:57d:78fb AAAA 2001:67c:1220:98a1:f124:24da:4d01:1914
1081 2.995218      fe80::c0ad:ac41:b2:1db9 ff02::fb        MDNS          97 Standard query 0x0000 A DOCK-SE6126.local, "QM" question
1084 3.075674      fe80::c0ad:ac41:b2:1db9 ff02::fb        MDNS          97 Standard query 0x0000 A DOCK-SE6126.local, "QM" question
1370 3.710749      fe80::c0ad:ac41:b2:1db9 ff02::fb        MDNS          111 Standard query response 0x0000 A 100.64.129.234
1412 3.806684      fe80::c0ad:ac41:b2:1db9 ff02::fb        MDNS          111 Standard query response 0x0000 A 100.64.129.234
1470 4.062726      fe80::c0ad:ac41:b2:1db9 ff02::fb        MDNS          97 Standard query 0x0000 A DOCK-SE6126.local, "QM" question
1475 4.106498      fe80::c0ad:ac41:b2:1db9 ff02::fb        MDNS          97 Standard query 0x0000 A DOCK-SE6126.local, "QM" question
2558 8.061204      fe80::c0ad:ac41:b2:1db9 ff02::fb        MDNS          97 Standard query 0x0000 A DOCK-SE6126.local, "QM" question
2560 8.061204      fe80::c0ad:ac41:b2:1db9 ff02::fb        MDNS          97 Standard query 0x0000 A DOCK-SE6126.local, "QM" question
3018 9.032959      fe80::c0ad:ac41:b2:1db9 ff02::fb        MDNS          97 Standard query 0x0000 A DOCK-SE6126.local, "QM" question
3020 9.113361      fe80::c0ad:ac41:b2:1db9 ff02::fb        MDNS          97 Standard query 0x0000 A DOCK-SE6126.local, "QM" question

+ Frame 460: 179 bytes on wire (1432 bits), 179 bytes captured (1432 bits) on 0
+ Ethernet II, Src: IntelGor_ff:b8:74 (8c:b8:7e:ff:b8:74), Dst: IPv6mcast_fb (33:33:00:00:00:fb)
+ Internet Protocol Version 6, Src: fe80::c0ad:ac41:b2:1db9, Dst: ff02::fb
+ User Datagram Protocol, Src Port: 5353, Dst Port: 5353
- Multicast Domain Name System (response)
  Transaction ID: 0x0000
  Flags: 0x8000 Standard query response, No error
  Questions: 0
  Answer RRs: 3
  Authority RRs: 0
  Additional RRs: 0
  Answers
    * LAPTOP-A48CIB6H.local: type AAAA, class IN, addr 2001:67c:1220:98a1:e34c:cd82:57d:78fb
    * LAPTOP-A48CIB6H.local: type AAAA, class IN, addr 2001:67c:1220:98a1:f124:24da:4d01:1914
    * LAPTOP-A48CIB6H.local: type AAAA, class IN, addr fe80::c0ad:ac41:b2:1db9
[Unsolicited: True]

```

Fig. 4.36: mDNS addresses not processed in the passive mode by the *ptnetinspector* – Wireshark output.

## 4.5.2 Timestamp addition

The one proposed change to the *ptnetinspector* is addition of the timestamps where the packet analysis started and finished so that the results of analysis are as accurate as possible. Originally, the time was obtained as the finish time of the *ptnetinspector* process, but it was not well synchronized. As individual modes perform analysis after capturing, there is certain timeout and the whole script finishes not strictly after the capturing is finished. This led to many non-analysed packets by the *ptnetinspector* and wrong results. After discussion with the supervisor, timestamps were implemented to the code of *ptnetinspector* and reported to the temporary file. This enabled the *Verification of results script* to compare each packet if it is within the range of analysed packets by the *ptnetinspector* and thus obtain correct results.

Originally captured end edited process end time is shown in the Fig. 4.37. As the process end time has precision to nanoseconds and captured packets have timestamps with precision to microseconds, function was written in the Python code (in the *Verification of results script*) to round the end time to microseconds.

The new version is displayed in the Fig. 4.38. It is the temporary file of the *ptnetinspector* that contains start time and end time of capturing (passive mode run for 5 seconds in the figure). Both time values are analysed inside the *Verification of results script* and the timestamp of every single packet is compared against these 2 values. If the packet timestamp is lower than the start time or higher than the end time, the packet is not analysed at all. This way the very accurate time synchronization is achieved.

The term "very accurate" is used intentionally as the synchronization is still not perfect. It was discovered that the timestamps are not accurate, specifically the end time stated by the *ptnetinspector* does not correspond to the timestamp of the last packet analysed. There is a difference in tens of milliseconds, which may lead (and also led – see the section Passive mode testing inside BUT network) to the mismatch in captured traffic and several packets can be omitted in the *ptnetinspector* analysis, but these packets appear in the *Verification of results script* analysis. In conclusion, the *Verification of results script* reports addresses (either MAC or IP) not discovered by the *ptnetinspector*.

```
1 05:15:57.103781830
2
```

Fig. 4.37: End time of the *ptnetinspector* process.

```
1 |time
2 2024-04-24 05:46:29.461835
3 2024-04-24 05:46:34.506258
4
```

Fig. 4.38: Start and end time of the *ptnetinspector* packet capturing.

### 4.5.3 Google public DNS address

When running aggressive mode where *ptnetinspector* proclaims the source machine (Kali) as a router, it was discovered that when Kali Linux sends a DNS query to the router which in turn responds with an answer, the source IP can be the public address of the Google DNS server (8.8.8.8). The *ptnetinspector* processes this result in a way that the address belongs to the router and assigns it to the particular MAC address (see the Fig. 4.39). The *Verification of results script* intentionally ignores this address, as can be seen in the Fig. 4.40.

This is a more general problem than just one related to Google DNS. Currently, *ptnetinspector* will proclaim every IP address coming from the internet as router's

```

[i] Device number 3: (Preferred router)
    MAC    ca:01:06:01:00:00
    IPv6   2001:f:b:a::1
    IPv6   2001:f:b:b::1
    IPv6   2001:f:b:c::1
    IPv6   2001:f:b:d::1
    IPv6   2001:f:b:e::1
    IPv4   8.8.8.8
    IPv6   fe80::c801:6ff:fe01:0
[i] Removing rules in configuration after scanning
[i] Manual IP configuration is not restored if being set
[i] Aggressive scan ended

```

Fig. 4.39: ptnetinspector Google DNS address assigned to the router.

```

##### OUTPUT OF PTNET
Device: 00:0c:29:0d:b2:e0
IP addresses: {'2001:f:b:b::a1', '2001:f:b:d:593e:cf0f:944a:b3d4', '2001:f:b:c:593e:cf0f:944a:b3d4', '169.254.179.212', 'fe80::593e:cf0f:944a:b3d4', '2001:f:b:e:593e:cf0f:944a:b3d4', '2001:f:b:a:e415:48c8:f55f:8d36', '2001:f:b:b:593e:cf0f:944a:b3d4', '2001:f:b:b:e415:48c8:f55f:8d36', '2001:f:b:a:593e:cf0f:944a:b3d4'}
Device: 00:50:56:c0:00:02
IP addresses: {'fe80::87ee:f1:2dc0:5ff', '192.168.191.1'}
Device: ca:01:06:01:00:00
IP addresses: {'8.8.8.8', 'fe80::c801:6ff:fe01:0', '2001:f:b:e::1', '2001:f:b:b::1', '2001:f:b:d::1', '2001:f:b:a::1', '2001:f:b:c::1'}
#####
Number of devices captured by the verifyAddresses script: 3
Number of devices captured by the verifyAddresses script: 18
###
Number of devices captured by the verifyAddresses script: 3
Number of devices captured by the verifyAddresses script: 19
#####
Every device and address captured was also found in the ptnet output.
#####
The address 8.8.8.8 was not found in the captured results for the ca:01:06:01:00:00 node.
Some addresses were not found or the set is empty.
#####

```

Fig. 4.40: Google address ignored in the verification script.

address. The suggestion for the future updates is to limit the reported results only to the local address range to avoid misleading information about addresses under which the router device is reachable.

## 4.6 Large-scale testing

In this section, large-scale testing is performed using the *Verification of results script*. Different approach is chosen for individual modes of the *ptnetinspector*. The passive mode does not probe the network for responses, it just idly listens to the incoming traffic, therefore such testing can be performed in the real BUT network. On the contrary, active and aggressive modes must be run in the environment of virtual machines. Testing these modes in the real network would lead into an excessive

number of packets generated and the security policies set by the local administrators might be violated. The source machine could be then added to the blacklist. All the following testings are performed after implementation of some of the proposed changes, specifically addition of timestamps and reading mDNS APDU in the passive mode.

#### 4.6.1 Passive mode testing inside BUT network

As already described, the passive mode is tested inside the real BUT Wi-Fi network (Wi-Fi eduroam). Dozens of devices and addresses communicate within this network, therefore it is a great environment to test both scripts under heavy load. The *Verification of results script* is executed with the following command, where the duration of listening is set to 30 seconds:

```
1 sudo ./verifyAddresses.py -i eth0 mode p -d 30
```

The command was executed 10 times during the noon. The runs were initiated with several seconds break. The results are displayed in the Tab. 4.6. The column values contain 2 numbers separated by the slash symbol. As shown in the column names, the (v/p) notation represents the result of the *Verification of results script* (v) and the result of *ptnetinspector* (p).

Table 4.6: Passive mode testing inside BUT network.

Test number	MACs found (v/p)	MACs missed (v/p)	IPs found (v/p)	IPs missed (v/p)
Test 1	113/113	0/0	247/247	0/0
Test 2	104/104	0/0	225/225	0/0
Test 3	120/120	0/0	253/253	0/0
Test 4	101/101	0/0	225/225	0/0
Test 5	103/102	0/1	212/211	0/1
Test 6	110/110	0/0	240/240	0/0
Test 7	115/115	0/0	230/230	0/0
Test 8	112/112	0/0	230/230	0/0
Test 9	100/100	0/0	199/199	0/0
Test 10	100/100	0/0	189/189	0/0

From the table Tab. 4.6, it can be clearly seen that the success rate is nearly 100 %. In the test number 5, there is a case where one device (MAC address) was

not detected together with its IP address by the *ptnetinspector*. The results after comparison are displayed in the Fig. 4.41. It can be seen that the device MAC address is *52:34:7d:20:96:98*. After further analysis, the *ptnetinspector* states that the end time of the capturing is 06:18:29.820998 (see the Fig. 4.42), and the first occurrence of the packet with such MAC address is at 06:18:29.789800 (according to the *Traffic capturing script*, see the Fig. 4.43). The packet should therefore be processed by the *ptnetinspector*. But after opening the temporary file of captured incoming packets by the *ptnetinspector*, it can be seen in the Fig. 4.44 that not only the MAC address is not present in the file, but also the timestamp of the last captured packet is 06:18:29.789799, therefore there is a 31.199 milliseconds gap. This led into more packets captured and processed by the *Verification of results script* and inconsistent results. The files are included in the Appendix D, specifically *DeviceNotCaptured* folder (or alternatively available on the public GitHub project). The output with the provided files can be verified in the console after running the *Verification of results script* with the *-nodet* flag.

```

kali@kali: ~/Documents/ptnetinspector_beta
File Actions Edit View Help
Device: f8:89:d2:b1:55:05
IP addresses: {'100.64.129.38', 'fe80::e617:cd12:9b9b:6c56'}
Device: f8:ff:c2:3d:c2:ef
IP addresses: {'fe80::1cba:ca3e:d590:10ca', '100.64.133.143'}
Device: fa:62:2c:a2:15:6e
IP addresses: {'fe80::1001:ccaf:421f:e91c', '100.64.135.220'}
Device: fc:d9:08:4d:f8:98
IP addresses: {'fe80::d2aa:e8f8:5e3f:7d04', '100.64.134.94', '2001:67c:1220:98a1:60f:2309:a74:45b7', '2001:67c:1220:98a1:5f9:7f37:682d:48c2'}
#####
#####
Number of devices captured by the verifyAddresses script: 103
Number of Addresses captured by the verifyAddresses script: 212
###
Number of devices captured by the ptnetinspector: 102
Number of Addresses captured by the ptnetinspector: 211
#####
#####
52:34:7d:20:96:98 was not captured by the ptnetinspector.
Some addresses were not found or the set is empty.
#####
#####
Every device and address from the ptnet output was also found in the captured results.
#####
#####
kali@kali:~/Documents/ptnetinspector_beta
$

```

Fig. 4.41: Passive mode testing results with 1 device missed by the ptnetinspector.

## 4.6.2 Active mode testing of various Windows 10 builds

For the purpose of active and aggressive mode testing, an array of Windows 10 Pro distributions was created. There are 12 versions, different build each. Their list is displayed in the Tab. 4.7. The purpose of this testing is to examine and observe

```

1 time
2 2024-04-26 06:17:59.768243
3 2024-04-26 06:18:29.820998
4

```

Fig. 4.42: Start and end timestamps stated by the ptnetinspector.

```

(kali@kali)-[~/Documents/ptnetinspector_beta]
└─$ cat CapturedPackets/ALL_Packets.txt | grep "52:34:7d:20:96:98 >"
06:18:29.789800 52:34:7d:20:96:98 > 01:00:5e:00:00:fb, ethertype IPv4 (0x0800), length 124: 100.64.131.176.5353 > 224.0.0.251.5353: 0 [3q] PTR (QU)? _companion-link._tcp.local. PTR (QU)? _rdlink._tcp.local. PTR (QU)? _sleep-proxy._udp.local. (82)
06:18:30.804022 52:34:7d:20:96:98 > 01:00:5e:00:00:fb, ethertype IPv4 (0x0800), length 124: 100.64.131.176.5353 > 224.0.0.251.5353: 0 [3q] PTR (QM)? _companion-link._tcp.local. PTR (QM)? _rdlink._tcp.local. PTR (QM)? _sleep-proxy._udp.local. (82)
06:18:30.804893 52:34:7d:20:96:98 > 33:33:00:00:00:fb, ethertype IPv6 (0x86dd), length 144: fe80::4eb:18b2:7c07:ff8f.5353 > ff02::fb.5353: 0 [3q] PTR (QM)? _companion-link._tcp.local. PTR (QM)? _rdlink._tcp.local. PTR (QM)? _sleep-proxy._udp.local. (82)
06:18:32.135578 52:34:7d:20:96:98 > 33:33:00:00:00:16, ethertype IPv6 (0x86dd), length 90: fe80::4eb:18b2:7c07:ff8f > ff02::16: HBH ICMP6, multicast listener report v2, 1 group record(s), length 28
(kali@kali)-[~/Documents/ptnetinspector_beta]
└─$

```

Fig. 4.43: Packets captured by the designed script with omitted source MAC address by the ptnetinspector.

```

2045 2024-04-26 06:18:29.576645,fc:d9:08:4d:f8:98,"Ether / IP / UDP / DNS Qry "b'Android-38.local.'" "
2046 2024-04-26 06:18:29.576646,fc:d9:08:4d:f8:98,"Ether / IPv6 / UDP / DNS Qry "b'Android-38.local.'" "
2047 2024-04-26 06:18:29.678184,68:54:5a:8c:44:eb,"Ether / IP / UDP / DNS Qry "b'_microsoft_mcc._tcp.local.'" "
2048 2024-04-26 06:18:29.781017,dc:68:0c:3f:d4:98,Ether / IPv6 / ICMPv6ND_RA / ICMPv6NDOptSrcLLAddr / ICMPv6NDOptMTU / ICMPv6NDOptRDNSS / ICMPv6NDOptDNSSL / ICMPv6 Neighbor Discovery Option - Prefix Information 2001:67c:1220:98a1::/64 On-link 1 Autonomous Address 1 Router Address 0
2049 2024-04-26 06:18:29.789799,fc:d9:08:4d:f8:98,"Ether / IP / UDP / DNS Qry "b'Android-38.local.'" "
2050

```

```

kali@kali: ~/Documents/ptnetinspector_beta
File Actions Edit View Help
(kali@kali)-[~/Documents/ptnetinspector_beta]
└─$ cat src/tmp/time_incoming.csv | grep "52:34:7d:20:96:98"
(kali@kali)-[~/Documents/ptnetinspector_beta]
└─$

```

Fig. 4.44: Actual packets captured by the ptnetinspector.

reactions of one of the most widely used operating systems among the end users to the IPv6 messages of various protocols. Network discovery feature is enabled on all the versions. The testing methodology involves running the machines four times in total. The scripts are executed three times consecutively, followed by the fourth run after restarting the machine.

The testing is performed in a way that only the Kali (source point of the scripts),



Table 4.7: Windows 10 Pro versions included in the active mode testing.

<b>Codename</b>	<b>Version</b>	<b>Build</b>
Threshold	1507	10240
Threshold 2	1511	10586
Redstone	1607	14393
Redstone 3	1709	16299
Redstone 5	1809	17763
19H1	1903	18362
19H2	1909	18363
20H1	2004	19041
20H2	20H2	19042
21H1	21H1	19043
21H2	21H2	19044
22H2	22H2	19045

router and the particular Windows 10 build are active in the topology to make the further analysis of the responses easier. A static global unicast address is assigned to each build from the  $2001:f:b:b::/64$  range, where the last byte consists of  $aX$  and  $X$  represents the order of the build (10240 is the first build, therefore the address is  $2001:f:b:b::a1$ ). The results can be seen in the Tab. 4.8.

As can be seen from the results, responses to some protocols are always the same (like positive responses to MLDv1 and LLMNR, not responding to the multicast ping, malicious ping and unknown ICMPv6 message of type 254) and some vary with different versions. It can be seen that the first 3 builds (versions 1507, 1511, 1607) send responses to the ICMPv6 queries in the default firewall state (private network profile). After exploring the firewall rules, the response rule is not active, but the builds still respond to all the set addresses. By default, ping was issued to the statically set GUA and L-L addresses. All later versions correctly follow the firewall rules and do not respond to the queries at all if it is not specifically set. Regarding the mDNS protocol, the first 4 versions tested do not send responses to the queries, even though the firewall rules exist for the mDNS and they are active in the default state. The responses are sent from the version 1809. Regarding the MLDv2 protocol, it was discovered that for all the tested versions, Windows responds with MLDv2 report messages to the MLD queries only during the first run of the scripts. If the scripts are run shortly after that, Windows responds only with the MLDv1 report messages. Apart from that, NS and NA conversations work without limitations, all of the Windows versions always react to the NS messages

and generate them themselves.

When running *ptnetinspector* repeatedly in a short time, it was also discovered that it irregularly sends / does not send the mDNS and LLMNR queries. In cases where queries are not sent, the number of output addresses detected is significantly lower (usually IPv4 address, Link-Local address, source addresses used for the communication and possible addresses derived from the MLD payload).

### 4.6.3 Active mode testing of various operating systems

In this section, testing of various operating systems (middle section machines in the Fig. 4.2) is performed. The operating systems used for the testing are displayed in the Tab. 4.9 together with their versions. The results of testing are displayed in the Tab. 4.10. Testing methodology is the same as in the case of Windows array testing.

Table 4.9: Various OS versions included in the active mode testing.

OS name	Version
Windows XP	Build 2600 (SP3)
Windows 7	Build 7601 (SP1)
Windows 11	Build 22621.1702 (22H2)
Ubuntu	22.04.3 LTS
Arch Linux	2024.03.01
macOS	Monterey 12.0.1
Android	Android-x86 9.0
Linux Mint	21.3 Virginia
openSUSE	Leap 15.0
CentOS	Stream release 9

It can be seen that Windows XP does not respond with MLDv2 report messages. Windows 7 follows the behavior (regarding MLD) of Windows 10 distributions. Neither of the 2 Windows versions responds to standard ICMPv6 queries in the firewall default state as well as to the LLMNR and mDNS queries. In case of Windows 11, when compared to other Windows distributions, it did not respond to LLMNR and mDNS queries originating from the Kali Linux. But regarding mDNS, it was captured that the responses without queries were sent during initial boot up after restart (see the Fig. 4.45). Similar behavior applies for the Ubuntu, Linux Mint, openSUSE and CentOS machines. They also send mDNS reports during initial boot up, but do not react to neither LLMNR nor mDNS queries sent from Kali. Android, Linux Mint and openSUSE send MLDv2 report messages during



the first run of scripts, but they respond only with MLDv1 reports after repetitive runs. Ubuntu, Arch Linux and CentOS always respond with MLDv2, no matter the run of scripts. All the Linux machines as well as macOS and Android respond with MLDv1 reports, send response to multicast ping and default ICMPv6 ping (directed to a specific address of the device – static GUA and L-L tested) and NS /NA messages. The unusual behavior was detected with the CentOS machine. From the tested Linux machines, CentOS was the only one who did not respond to ICMPv6 multicast messages with the set unknown type (254). All of the Linux machines together with macOS and Android responded to malicious multicast pings, Android responded also to the unicast ICMPv6 unknown message.

1115 39.518860	fe80::e734:2b41:5223:b9c9	ff02::fb	MDNS	101 Standard query 0x0000 ANY DESKTOP-10E2378.local, "QM" question
1116 39.519582	fe80::e734:2b41:5223:b9c9	ff02::fb	MDNS	431 Standard query response 0x0000 AAAA 2001:fb:a:21d0:d656:c50a:2a8 AAAA 2001:fb:b
1117 39.520494	fe80::e734:2b41:5223:b9c9	ff02::1:3	LLMNR	95 Standard query 0xec3e ANY DESKTOP-10E2378
1139 41.283474	192.168.0.5	224.0.0.251	MDNS	81 Standard query 0x0000 ANY DESKTOP-10E2378.local, "QM" question
1140 41.283492	192.168.0.5	224.0.0.251	MDNS	427 Standard query response 0x0000 AAAA 2001:fb:a:21d0:d656:c50a:2a8 AAAA 2001:fb:b
1141 41.283538	fe80::e734:2b41:5223:b9c9	ff02::fb	MDNS	101 Standard query 0x0000 ANY DESKTOP-10E2378.local, "QM" question
1142 41.283556	fe80::e734:2b41:5223:b9c9	ff02::fb	MDNS	447 Standard query response 0x0000 AAAA 2001:fb:a:21d0:d656:c50a:2a8 AAAA 2001:fb:b
1143 41.283598	fe80::e734:2b41:5223:b9c9	ff02::1:3	LLMNR	95 Standard query 0x25d1 ANY DESKTOP-10E2378
1144 41.283615	192.168.0.5	224.0.0.252	LLMNR	75 Standard query 0x25d1 ANY DESKTOP-10E2378
1151 41.364191	192.168.0.5	224.0.0.251	MDNS	81 Standard query 0x0000 ANY DESKTOP-10E2378.local, "QM" question
1152 41.364777	192.168.0.5	224.0.0.251	MDNS	427 Standard query response 0x0000 AAAA 2001:fb:a:21d0:d656:c50a:2a8 AAAA 2001:fb:b
1153 41.364922	fe80::e734:2b41:5223:b9c9	ff02::fb	MDNS	101 Standard query 0x0000 ANY DESKTOP-10E2378.local, "QM" question
1154 41.365054	fe80::e734:2b41:5223:b9c9	ff02::fb	MDNS	447 Standard query response 0x0000 AAAA 2001:fb:a:21d0:d656:c50a:2a8 AAAA 2001:fb:b
1155 41.365919	fe80::e734:2b41:5223:b9c9	ff02::1:3	LLMNR	95 Standard query 0xa33e ANY DESKTOP-10E2378
1156 41.365935	192.168.0.5	224.0.0.252	LLMNR	75 Standard query 0xa33e ANY DESKTOP-10E2378

Fig. 4.45: LLMNR and mDNS messages generated by Windows 11 during booting.

#### 4.6.4 Aggressive mode testing of various operating systems

For the purpose of aggressive mode testing, multiple instances of Windows 10 operating systems and Linux machines are run. Specifically, the machines are Windows 11, Windows 10 22H2, Windows 10 1809, Ubuntu, Arch Linux and CentOS. Similar approach is taken as in the case of passive mode testing, i.e. number of captured devices and addresses. Aggressive mode is run using the following command:

```
1 sudo ./verifyAddresses.py -i eth0 mode a+ -da+ 30
```

Testing methodology is similar to the passive testing, 10 tests are performed consecutively. The results are shown in the Tab. 4.11. The success rate of testing is nearly 100 %. Even though the outputs of tests 3 and 5 tell that one IP address was missed by the *verifyAddresses* script, it is the public DNS address of Google that is intentionally ignored (as discussed in the section 4.5.3). Such on output of Test 5 is shown in the Fig. 4.46. The files are included in the Appendix D, specifically *GoogleIP* folder (or alternatively available on the public GitHub project). The same address problem applies for the Test 3.

```
#####
#####
Number of devices captured by the verifyAddresses script: 8
Number of Addresses captured by the verifyAddresses script: 67
###
Number of devices captured by the ptnetinspector: 8
Number of Addresses captured by the ptnetinspector: 68
#####

#####
Every device and address captured was also found in the ptnet output.
#####

#####
The address 8.8.8.8 was not found in the captured results for the ca:01:06:01:00:00 node.
Some addresses were not found or the set is empty.
#####
```

Fig. 4.46: Aggressive mode testing – ignored Google DNS address by the designed script.

Table 4.11: Aggressive mode testing with virtual machines.

Test number	MACs found (v/p)	MACs missed (v/p)	IPs found (v/p)	IPs missed (v/p)
Test 1	8/8	0/0	64/64	0/0
Test 2	8/8	0/0	64/64	0/0
Test 3	8/8	0/0	62/63	1/0
Test 4	8/8	0/0	63/63	0/0
Test 5	8/8	0/0	67/68	1/0
Test 6	8/8	0/0	62/62	0/0
Test 7	8/8	0/0	61/61	0/0
Test 8	8/8	0/0	66/66	0/0
Test 9	8/8	0/0	63/63	0/0
Test 10	8/8	0/0	62/62	0/0

#### 4.6.5 Scanning vulnerability

From the point of view of scanning the devices, Windows machines have quite repetitive behavior. Simply by observing the reactions of systems to the IPv6 flows, Windows 10 can be characterized by their response to the LLMNR, and later versions to the mDNS protocols. Linux machines, on the other hand, respond to unknown ICMPv6 messages except for the CentOS. All the Linux machines together with macOS and Android respond to malicious ping and regular multicast ping.

Table 4.8: Windows 10 various build responses – active mode (default firewall rules).

Windows 10 version	MLDv2 <sup>a</sup>	MLDv1	Multicast ping	Unknown ICMPv6 (254)	Malicious ping	Default ICMPv6 (ping)	LLMNR	mDNS	NS
1507	✓/✗	✓	✗	✗	✗	✓	✓	✗	✓
1511	✓/✗	✓	✗	✗	✗	✓	✓	✗	✓
1607	✓/✗	✓	✗	✗	✗	✓	✓	✗	✓
1709	✓/✗	✓	✗	✗	✗	✗	✓	✗	✓
1809	✓/✗	✓	✗	✗	✗	✗	✓	✓	✓
1903	✓/✗	✓	✗	✗	✗	✗	✓	✓	✓
1909	✓/✗	✓	✗	✗	✗	✗	✓	✓	✓
2004	✓/✗	✓	✗	✗	✗	✗	✓	✓	✓
20H2	✓/✗	✓	✗	✗	✗	✗	✓	✓	✓
21H1	✓/✗	✓	✗	✗	✗	✗	✓	✓	✓
21H2	✓/✗	✓	✗	✗	✗	✗	✓	✓	✓
22H2	✓/✗	✓	✗	✗	✗	✗	✓	✓	✓

<sup>a</sup>The stations send MLDv2 response only during the first run of scripts, then respond only with MLDv1. This is the reason why the check mark, together with the cross, is written down.

Table 4.10: Various OS responses – active mode (default firewall rules).

OS	MLDv2 <sup>a</sup>	MLDv1	Multicast ping	Unknown ICMPv6 (254)	Malicious ping	Default ICMPv6 (ping)	LLMNR	mDNS <sup>b</sup>	NS
Windows XP	✗	✓	✗	✗	✗	✗	✗	✗	✓
Windows 7	✓/✗	✓	✗	✗	✗	✗	✗	✗	✓
Windows 11	✓/✗	✓	✗	✗	✗	✗	✗	✓/✗	✓
macOS	✗	✓	✓	✓	✓	✓	✗	✗	✓
Ubuntu	✓	✓	✓	✓	✓	✓	✗	✓/✗	✓
Arch Linux	✓	✓	✓	✓	✓	✓	✗	✗	✓
Android	✓/✗	✓	✓	✓	✓	✓	✗	✓	✓
Linux Mint	✓/✗	✓	✓	✓	✓	✓	✗	✓/✗	✓
openSUSE	✓/✗	✓	✓	✓	✓	✓	✗	✓/✗	✓
CentOS	✓	✓	✓	✗	✓	✓	✗	✓/✗	✓

<sup>a</sup>The same scenario as in case of various Windows 10 builds.

<sup>b</sup>Some devices do not send response to the queries sent from the *ptnetinspector*, but during the booting process, it can be seen that devices send their reports on their own. Therefore, both the check mark and cross symbols are written again.

# Conclusion

The master's thesis aim was to study the theoretical IPv6 concepts, establish testing environment for the proposed scenarios, develop scripts for the *ptnetinspector* tool verification and discover differences between various operating systems regarding their IPv6 implementation.

For the testing environment, the GNS3 emulator was chosen. The instances of devices primarily run on the local server and are interconnected by the switches and a Cisco router. Kali runs the *ptnetinspector* program which intercepts or even generates the traffic according to one of its 3 modes (passive, active, aggressive). Each of the modes was tested against 5 proposed scenarios. The scenarios were designed in a way that resource utilization for the tool execution was measured with increasing load, i.e. higher number of addresses each device possesses. The Bash script was developed for the automated measuring. The results for the active and aggressive modes are shown in the Tab. 4.4 and Tab. 4.5 respectively.

Another Bash script was developed for the traffic capturing together with Python script which compares own detected devices and their respective addresses with the results of *ptnetinspector*. Certain shortcomings (as described in the chapter 4.5) were discovered, i.e. *ptnetinspector* did not analyze payload of the mDNS packets during passive mode scanning and reports public addresses as belonging to the router device (based on the MAC address). Additionally, implementation of timestamps was proposed to better synchronize the traffic capturing of all scripts.

After successfully implementing the Python verification script, it was used to observe the responses of various operating systems to different IPv6 flows. All 3 modes were tested from different perspectives. As can be seen in the Tab. 4.6 (passive testing) and Tab. 4.11 (aggressive testing), the results of both scripts corresponded nearly 100%, but the synchronization of *ptnetinspector* is still not perfect and it still does not deal with public addresses. Various Windows 10 builds (Tab. 4.8) and other operating systems (mainly Linux – Tab. 4.10) were tested in active mode. Responses of Windows 10 are pretty similar across successive builds, they respond to MLDv1, MLDv2, LLMNR and NS messages. The mDNS reporting starts from the version 1809. First 3 versions respond to ICMPv6 query messages by default. None of the Linux distributions responds to LLMNR queries or reacts to mDNS queries sent by the *ptnetinspector*, but mDNS protocol is implemented and reports sent during booting can be captured for macOS, Linux Mint, openSUSE and CentOS. Android always responds to mDNS queries. The only Linux machine not responding to the ICMPv6 query type 254 is CentOS. Android, macOS and Linux distributions respond to the malicious and multicast ping, which differs them from Windows.

The developed scripts are attached in the ZIP archive (see Appendix D).

RQ1: *Are there any detection errors or other bugs in `ptnetinspector` tool provided for IPv6 devices discovery? Is there a way to resolve them?*

In the section 4.5.1, the problem of `ptnetinspector` tool not reading the mDNS messages during passive mode scanning, respectively their payload containing device addresses, was described. After reporting the problem to the author of the application, it was fixed. Another problem that arose during the verification process was synchronization between the proposed script and `ptnetinspector`. Originally the timestamp of the `ptnetinspector` process finish time was captured, but it was not accurate. After reporting this to the author, timestamps were implemented to the code, but as already explained in the section 4.5.2, the synchronization is still not perfect as the reported end timestamp does not correspond to the processed packets (slight delay in order of tens of milliseconds). The last error found was `ptnetinspector` reporting public addresses as belonging to the local router device. This was shown in the section 4.5.3, where the public Google DNS address was printed under the router's addresses. Therefore, all discovered errors are already fixed or could be fixed quite easily. During large-scale testing of `ptnetinspector` tool, no other errors or design flaws were observed.

RQ2: *What is the performance of `ptnetinspector` tool under certain scenarios?*

As can be seen in the Tab. 4.4 for active scanning, with the increasing load of input data (higher number of addresses per device), the average runtime grows larger. The change is most apparent from Scenario 4, where router starts distributing prefixes. Under small load, the `ptnetinspector` runs around 11 seconds, but with a high number of addresses, it can take over half a minute. CPU usage did not exceed 50 % of 1 core. It can be clearly seen that higher input load leads to more RAM usage. In case of no IPv6 device in the topology, RAM usage was approximately 98 MB. For the 5 distributed prefixes by the router scenario, it was approximately 114 MB. During the aggressive scanning (Tab. 4.5), similar results with increasing demands on the computational resources can be observed. For the no IPv6 device scenario, the average runtime is 31.75 seconds, while for the most demanding scenario the average runtime is 135.658 seconds. The average CPU usage is quite low (around 16 %), but as discussed in the section 4.3.1, the load is spread across larger time interval, where there are short peaks in high CPU utilization, but then there are long periods of inactivity

(the tool just passively listens to the traffic). In terms of RAM usage, the results are quite comparable with active mode, the lowest value is approximately 91 MB and the highest approximately 108 MB. But the results are again influenced by long inactivity periods. The results also depend on the addresses the devices report or use for the communication. Therefore, the *ptnetinspector* could be run easily, there are no significant constraints in terms of computational or other resources. The duration of test is dependent on the particular conditions with expected dependency.

RQ3: *What, if any, are the differences between the selected major operating systems in terms of specific aspects of IPv6 implementation?*

From the tested Windows 10 VM array (Tab. 4.8), it can be observed that various builds have quite repetitive behavior. All the tested builds respond to MLDv1 and LLMNR. MLDv2 reports are sent only during the first run of scripts. On the contrary, they do not respond to the multicast ping, unknown ICMPv6 message (type 254) and malicious ping (unknown option 128). First 3 tested versions (1507, 1511 and 1607) respond to ICMPv6 queries in a default firewall state (private profile) with disabled rules for the response and mDNS reports are sent from the version 1809. Windows 11 follows similar behavior but does not send LLMNR responses. In the Tab. 4.10, where among 3 other Windows version the various Linux distributions, macOS and Android device were tested, quite different results can be observed. None of the machines responds to LLMNR. All the Linux distributions with macOS and Android respond to multicast ping, malicious ping and ICMPv6 query messages. CentOS is the only Linux distribution that does not respond to the ICMPv6 message of type 254. Ubuntu, Arch Linux and CentOS send together with MLDv1 also MLDv2 reports, no matter the script run. macOS neither sends MLDv2 nor mDNS responses. Out of the tested VMs, Android is the only device that answers mDNS queries. Ubuntu, Linux Mint, openSUSE and CentOS support mDNS protocol, but do not respond to the *ptnetinspector* queries. Therefore, it was confirmed there are several differences in IPv6 default behavior between selected major operating systems. Of course, these differences could be studied in more details and other aspects, such as different firewall profiles or antivirus softwares could be taken into account in possible follow-up works.

As obvious from the above text, all three research questions were resolved successfully.

# Bibliography

- [1] IBM. *What is virtualization?* Online. IBM. 2019. Available at: <https://www.ibm.com/topics/virtualization>. [cit. 2023-10-07].
- [2] KOMOSNÝ, Dan. *Síťové operační systémy*. Skriptum FEKT Vysoké učení technické v Brně, 2022 [cit. 2023-10-07]. s. 1-129.
- [3] SIMIC, Sofija. *What is a Hypervisor? Types of Hypervisors 1 & 2*. Online. PhoenixNAP. 2022. Available at: <https://phoenixnap.com/kb/what-is-hypervisor-type-1-2>. [cit. 2023-10-07].
- [4] VMWARE. *What is network virtualization?* Online. 2023. Available at: <https://www.vmware.com/topics/glossary/content/network-virtualization.html>. [cit. 2023-10-07].
- [5] GNS3 COMMUNITY. *Getting Started with GNS3*. Online. GNS3. 2020. Available at: <https://docs.gns3.com/docs/>. [cit. 2023-09-24].
- [6] GROSSMANN, Jeremy. *GNS3 2.2.36 Released!*. Online. GNS3. 2023. Available at: <https://gns3.com/community/blog/gns3-2-2-36-released>. [cit. 2024-01-07].
- [7] Paolo (paolo-projects). *Unlocker 3.0.5*. Online. GitHub. 2023. Available at: <https://github.com/paolo-projects/unlocker/releases>. [cit. 2024-03-19].
- [8] HUANG, Chih-Wei and others. *Android-x86*. Online. Android-x86. 2022. Available at: <https://www.android-x86.org/>. [cit. 2024-03-19].
- [9] SHARMA, Sonal. *Alpha Testing vs Beta Testing: Everything you need to know*. Online. Testsigma. 2023. Available at: <https://testsigma.com/blog/alpha-test-vs-beta-test/>. [cit. 2023-11-04].
- [10] HAMILTON, Thomas. *Manual Testing Tutorial: What is, Types, Concepts*. Online. Guru99. 2023. Available at: <https://www.guru99.com/manual-testing.html>. [cit. 2023-11-04].



- [11] HAMILTON, Thomas. *What is Automation Testing? Test Tutorial*. Online. Guru99. 2023. Available at:  
<https://www.guru99.com/automation-testing.html>. [cit. 2023-11-04].
- [12] Inflectra. *Functional vs. Non-Functional Testing Methodologies*. Online. Inflectra. 2023. Available at:  
<https://www.inflectra.com/Ideas/Topic/Functional-vs-Non-Functional-Testing.aspx>. [cit. 2023-11-04].
- [13] SmartBear Software. *Software Testing Methodologies*. Online. SmartBear Software. 2023. Available at:  
<https://smartbear.com/learn/automated-testing/software-testing-methodologies/>. [cit. 2023-11-04].
- [14] CHATTERJEE, Shormistha. *What is Test Methodology? (With 7 Methodologies)*. Online. BrowserStack. 2022. Available at:  
<https://www.browserstack.com/guide/software-testing-methodologies>. [cit. 2023-11-04].
- [15] Inflectra. *Software Testing Methodologies - Learn the Methods & Tools*. Online. Inflectra. 2023. Available at:  
<https://www.inflectra.com/Ideas/Topic/Testing-Methodologies.aspx>. [cit. 2023-11-04].
- [16] HAMILTON, Thomas. *Software Testing Methodologies: QA Models*. Online. Guru99. 2023. Available at:  
<https://www.guru99.com/testing-methodology.html>. [cit. 2023-11-04].
- [17] DEERING, S. and HINDEN, R. *Internet Protocol, Version 6 (IPv6) Specification*. Online. RFC 2460. 10.17487/RFC2460, 1998. Available at:  
<https://doi.org/10.17487/RFC2460>. [cit. 2023-10-08].
- [18] JEŘÁBEK, J. *Pokročilé komunikační techniky*. Skriptum FEKT Vysoké učení technické v Brně, 2023. s. 1-180. [cit. 2023-10-08].
- [19] Network Academy. *IPv4 vs IPv6 - Understanding the differences*. Online. Network Academy. 2023. Available at:  
<https://www.networkacademy.io/ccna/ipv6/ipv4-vs-ipv6>. [cit. 2023-10-28].

- [20] Cisco. *IPv6 MTU Path Discovery*. Online. Cisco. 2023. Available at: [https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/ipv6\\_basic/configuration/xe-3s/ip6b-xe-3s-book/ip6-mtu-path-disc.pdf](https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/ipv6_basic/configuration/xe-3s/ip6b-xe-3s-book/ip6-mtu-path-disc.pdf). [cit. 2023-10-28].
- [21] Imperva. *Anycast*. Online. Imperva. 2023. Available at: <https://www.imperva.com/learn/performance/anycast/>. [cit. 2023-10-29].
- [22] HINDEN, R. and DEERING, S. *IP Version 6 Addressing Architecture*. Online. RFC 4291. 10.17487/RFC4291, 2006. Available at: <https://doi.org/10.17487/RFC4291>. [cit. 2023-10-29].
- [23] Cisco Press. *IPv6 Address Representation and Address Types*. Online. Cisco Press. 2017. Available at: <https://www.ciscopress.com/articles/article.asp?p=2803866&seqNum=4>. [cit. 2023-10-29].
- [24] KHANNA, Sunil. *Understanding IPv6 EUI-64 Bit Address*. Online. Cisco Community. 2012. Available at: <https://community.cisco.com/t5/networking-knowledge-base/understanding-ipv6-eui-64-bit-address/ta-p/3116953>. [cit. 2023-10-07].
- [25] CONTA, A.; DEERING, S. and GUPTA, M. *Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification*. Online. RFC 4443. 10.17487/RFC4443, 2006. Available at: <https://doi.org/10.17487/RFC4443>. [cit. 2023-10-12].
- [26] SATRAPA, Pavel. *IPv6: internetový protokol verze 6*. 4. aktualizované a rozšířené vydání. CZ.NIC. Praha: CZ.NIC, 2019. ISBN 978-80-88168-46-1.
- [27] NARTEN, T.; NORDMARK, E.; SIMPSON, W. and SOLIMAN, H. *Neighbor Discovery for IP version 6 (IPv6)*. Online. RFC 4861. 10.17487/RFC4861, 2007. Available at: <https://doi.org/10.17487/RFC4861>. [cit. 2023-11-15].
- [28] DEERING, S.; FENNER, W. and HABERMAN, B. *Multicast Listener Discovery (MLD) for IPv6*. Online. RFC 2710. DOI 10.17487/RFC2710, 1999. Available at: <https://doi.org/10.17487/RFC2710>. [cit. 2023-12-02].
- [29] VIDA, R. and COSTA, L. *Multicast Listener Discovery Version 2 (MLDv2) for IPv6*. Online. RFC 3810. DOI 10.17487/RFC3810, 2004. Available at: <https://doi.org/10.17487/RFC3810>. [cit. 2023-12-02].

- [30] ANTHONY, Sebastian. *Who actually develops Linux? The answer might surprise you.* Online. ExtremeTech. Available at: <https://www.extremetech.com/computing/175919-who-actually-develops-linux-the-answer-might-surprise-you>. [cit. 2024-01-08].
- [31] PALLAS, Volker D. *IPv6 on Mac OS / The Definitive Guide.* Online. PALLAS DIGITAL. 2023. Available at: <https://pall.as/ipv6-on-mac/>. [cit. 2023-12-06].
- [32] APOORVA. *MacOS vs Linux: Key Differences that You Should Know.* Online. Scaler Topics. 2023. Available at: <https://www.scaler.com/topics/linux-vs-mac/>. [cit. 2023-12-06].
- [33] PHAN, Viet Anh. *Generování IPv6 a ICMPv6 paketů a jejich vliv na fungování zařízení v síti.* Online. Master's thesis. Brno: Brno University of Technology. 2023. Available at: <https://theses.cz/id/248pvf/>. [cit. 2023-12-06].

# Symbols and abbreviations

<b>VM</b>	Virtual machine
<b>CPU</b>	Central Processing Unit
<b>RAM</b>	Random Access Memory
<b>HDD</b>	Hard Disk Drive
<b>SSD</b>	Solid State Drive
<b>OS</b>	Operating System
<b>KVM</b>	Kernel-based Virtual Machine
<b>VLAN</b>	Virtual Local Area Network
<b>VPN</b>	Virtual Private Network
<b>IT</b>	Information Technology
<b>GPU</b>	Graphics Processing Unit
<b>GNS3</b>	Graphical Network Simulator-3
<b>GUI</b>	Graphical User Interface
<b>VPC</b>	Virtual Personal Computer
<b>TPM</b>	Trusted Platform Module
<b>IPv6</b>	Internet Protocol Version 6
<b>IoT</b>	Internet of Things
<b>NAT</b>	Network Address Translation
<b>QoS</b>	Quality of Service
<b>DSCP</b>	Differentiated Services Code Point
<b>ECN</b>	Explicit Congestion Notification
<b>TCP</b>	Transmission Control Protocol
<b>UDP</b>	User Datagram Protocol
<b>ToS</b>	Type of Service

<b>MTU</b>	Maximum Transmission Unit
<b>CDN</b>	Content Delivery Network
<b>DNS</b>	Domain Name System
<b>IANA</b>	Internet Assigned Numbers Authority
<b>RIR</b>	Regional Internet Registry
<b>LIR</b>	Local Internet Registry
<b>EUI-64</b>	Extended Unique Identifier
<b>IEEE</b>	Institute of Electrical and Electronics Engineers
<b>MAC</b>	Media Access Control
<b>OUI</b>	Organizational Unique Identifier
<b>NIC</b>	Network Interface Controller
<b>ICMPv6</b>	Internet Control Message Protocol version 6
<b>ND</b>	Neighbor Discovery
<b>ARP</b>	Address Resolution Protocol
<b>DHCP</b>	Dynamic Host Configuration Protocol
<b>SLAAC</b>	Stateless Address Autoconfiguration
<b>DUID</b>	DHCP Unique Identifier
<b>IA</b>	Identity Association
<b>MLD</b>	Multicast Listener Discovery
<b>IGMP</b>	Internet Group Management Protocol
<b>RFC</b>	Request for Comments
<b>ISP</b>	Internet Service Provider
<b>CMD</b>	Command Prompt
<b>CLI</b>	Command Line Interface
<b>APIPA</b>	Automatic Private IP Addressing

<b>LLMNR</b>	Link-Local Multicast Name Resolution
<b>OOD</b>	Object-Oriented Design
<b>APDU</b>	Application Protocol Data Unit

## List of appendices

A	Verification of results script code sample	106
B	Performance testing script code sample	108
C	Traffic capturing script code sample	109
D	Content of the electronic attachment	110

## A Verification of results script code sample

This Appendix contains part of the source code of the script designed for the purpose of extracting addresses and comparing results. The code below lists the function extracting addresses from the payload of LLMNR packets.

Listing A.1: Verification of results Python script code sample.

```
1 # Function to extract addresses from the LLMNR packets (  
    payload)  
2 def extractLLMNRaddresses(nodeArr):  
3     global uniqDevObjCapt  
4     macAddr = extractMac(nodeArr[0])  
5     lineWithBytes = r'^(\s*)0x[0-9a-z]{4}'  
6     bytesPattern = r'(?:[0-9a-f]{4}|[0-9a-f]{2})(?=\s)'  
7     extractedBytes = []  
8  
9  
10    for line in nodeArr:  
11        if re.match(lineWithBytes, line):  
12            lineBytes = re.findall(bytesPattern, line)  
13            for i in range(len(lineBytes)):  
14                extractedBytes.append(lineBytes[i])  
15  
16    for id, byte in enumerate(extractedBytes):  
17        try:  
18            # PTR record  
19            if (byte == '000c'):  
20                if((extractedBytes[id+1] == '0001') and (  
                    extractedBytes[id+2] == '0000') and (  
                        extractedBytes[id+3] == '001e')):  
21                    continue  
22            # A (IPv4) record  
23            elif (byte == '0001'):  
24                if((extractedBytes[id+1] == '0001') and (  
                    extractedBytes[id+2] == '0000') and (  
                        extractedBytes[id+3] == '001e')):  
25                    ipv4addrLen = int(int(extractedBytes[id+4], 16)  
                        /2)  
26                    ipv4addr = str()
```



```

27     for i in range(ipv4addrLen):
28         temp2Bytes = extractedBytes[id+5+i]
29         byteArr = [int(temp2Bytes[j:j+2],16) for j in
30                     range(0, len(temp2Bytes), 2)]
31         for num in byteArr:
32             ipv4addr += str(num) + '.'
33         ipv4addr = ipv4addr[:-1]
34         if(findDeviceByMAC(macAddr)):
35             uniqDevObjCapt.get(macAddr).set_ip_addresses(
36                 ipv4addr)
37     # AAAA (IPv6) record
38     elif(byte == '001c'):
39         if((extractedBytes[id+1] == '0001') and (
40             extractedBytes[id+2] == '0000') and (
41             extractedBytes[id+3] == '001e')):
42             ipv6addrLen = int(int(extractedBytes[id+4], 16)
43                               /2)
44             ipv6addr = str()
45             for i in range(ipv6addrLen):
46                 ipv6addr += extractedBytes[id+5+i].lstrip("0"
47                     ) + ':'
48             ipv6addr = re.sub(':{2,}', '::', ipv6addr)
49             ipv6addr = ipv6addr[:-1]
50             if(findDeviceByMAC(macAddr)):
51                 uniqDevObjCapt.get(macAddr).set_ip_addresses(
52                     ipv6addr)
53     except IndexError:
54         continue

```

## B Performance testing script code sample

Here is an example of the source code of the Bash script created for the purpose of performance testing. Function to calculate average run time is presented.

Listing B.1: Performance testing Bash script code sample.

```
1 ##### AVERAGE RUN TIME
2 realTime=false
3 runTimeArray=()
4 regex=' [0-9]+\.[0-9]+'
5 for line in $(cat ptnettime.txt)
6 do
7     if [ "$realTime" = true ];
8     then
9         runtime_sec=$(echo "$line" | grep -oE "$regex")
10        runtime_min=$(echo "$line" | grep -oE '^[0-9]+')
11        if [ -n "$runtime_min" ]; then
12            runtime_sec=$(echo "$runtime_sec" + "$((runtime_min
13                *60))" | bc)
14        fi
15        array+=($runtime_sec)
16        realTime=false
17        fi
18        if [[ $line =~ ^[real] ]];
19        then
20            realTime=true
21        fi
22    done
23    sumRT=0
24
25    for elm in "${array[@]}; do
26        sumRT=$(echo "$sumRT+$elm" | bc)
27    done
28
29    averageRT=$(echo "scale=3; ${sumRT}/${#array[@]}" | bc)
```

## C Traffic capturing script code sample

In this part, an example of the source code of the Bash script that captures network traffic is presented. The code below demonstrates extraction of the packets to separate files.

Listing C.1: Traffic capturing Bash script code sample.

```
1 # Read all the packets (not src MAC addr 00:0c:29:b8:a9
   :4d) to the text file
2 tcpdump -r ./CapturedPackets/capPackets.pcapng not ether
   src $3 -e -n > ./CapturedPackets/ALL_Packets.txt 2> ./
   CapturedPackets/tcpdump_stderr.log
3
4 echo -e "\nReading_MLD_packets..."
5 # Read all the MLD report packets
6 tcpdump -r ./CapturedPackets/capPackets.pcapng not ether
   src $3 and ip6 and not icmp6 and not udp port 5353 and
   not udp port 5355 -e -v -n > ./CapturedPackets/
   MLD_report_Packets.txt 2>> ./CapturedPackets/
   tcpdump_stderr.log
7
8 echo -e "\nReading_MDNS_packets..."
9 # Read all the MDNS packets
10 tcpdump -r ./CapturedPackets/capPackets.pcapng not ether
   src $3 and udp port 5353 -e -v -n > ./CapturedPackets/
   MDNS_Packets.txt 2>> ./CapturedPackets/tcpdump_stderr.
   log
11
12 echo -e "\nReading_LLMNR_packets..."
13 # Read all the LLMNR packets
14 tcpdump -r ./CapturedPackets/capPackets.pcapng not ether
   src $3 and udp port 5355 -e -s0 -vvv -X -n > ./
   CapturedPackets/LLMNR_Packets.txt 2>> ./
   CapturedPackets/tcpdump_stderr.log
```

## D Content of the electronic attachment

All the scripts are submitted in the ZIP archive *Ruiner\_attachment.zip*. Its structure can be seen below.

```
/.-----root of the attached archive
├── DeviceNotCaptured..... Wrong time synchronization by ptnetinspector
│   ├── CapturedPackets
│   │   ├── ALL_Packets.txt
│   │   ├── capPackets.pcapng
│   │   ├── CompareResults.txt
│   │   ├── LLMNR_Packets.txt
│   │   ├── MDNS_Packets.txt
│   │   ├── MLD_report_Packets.txt
│   │   ├── ptnetinspector_stderr.log
│   │   ├── ptnetOut.txt
│   │   ├── tcpdump_stderr.log
│   │   └── tcpdump_stdout.log
│   └── tmp
│       ├── start_end_mode.csv
│       └── time_incoming.csv
├── GoogleIP..... Public Google DNS IP proclaimed as router's IP
│   ├── CapturedPackets
│   │   ├── ALL_Packets.txt
│   │   ├── capPackets.pcapng
│   │   ├── CompareResults.txt
│   │   ├── LLMNR_Packets.txt
│   │   ├── MDNS_Packets.txt
│   │   ├── MLD_report_Packets.txt
│   │   ├── ptnetinspector_stderr.log
│   │   ├── ptnetOut.txt
│   │   ├── tcpdump_stderr.log
│   │   └── tcpdump_stdout.log
│   └── tmp
│       ├── start_end_mode.csv
│       └── time_incoming.csv
├── verifyAddresses.py..... Verification of results script
├── ptnetPerf.sh..... Performance testing script
└── captPackets.sh..... Traffic capturing script
```