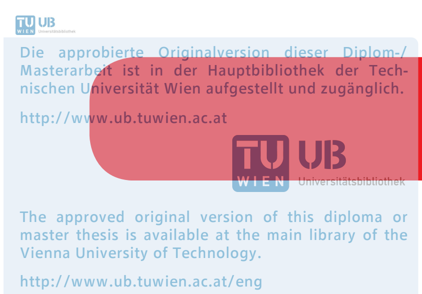

Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

IPv6 High Performance Scanning

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Business Informatics

eingereicht von

Christoph Kukovic, BSc
Matrikelnummer 1025759

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: PD Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl
Mitwirkung: Dipl.-Ing. Johanna Ullrich, BSc

Wien, 21. August 2016

Christoph Kukovic Edgar Weippl

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

IPv6 High Performance Scanning

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

Christoph Kukovic, BSc
Registration Number 1025759

to the Faculty of Informatics

at the TU Wien

Advisor: PD Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl
Assistance: Dipl.-Ing. Johanna Ullrich, BSc

Vienna, 21st August, 2016

Christoph Kukovic Edgar Weippl

Erklärung zur Verfassung der Arbeit

Christoph Kukovic, BSc
Schubertgasse 10, 1090 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 21. August 2016

____

Christoph Kukovic

Kurzfassung

Die Einführung des neuen Internetprotokolls IPv6 erfordert eine umfangreiche Umstellungsphase die potenzielle Sicherheitslücken mit sich bringt. Da sowohl für private Personen und Unternehmen als auch für den öffentlichen Sektor IT-Security eine immer wichtigere Rolle spielt, gilt es, die Gefahren dieser Phase rechtzeitig aufzuzeigen und zu eliminieren. In der Vergangenheit bot IP High Performance Scanning einen effizienten Weg, um die globale Internetinfrastruktur zu analysieren und potenzielle Gefahrenpunkte zu identifizieren. Da das Vorgängerprotokoll von IPv6, IPv4 eine mittlerweile überschaubare Anzahl an verfügbaren, eindeutigen Netzwerkadressen bietet, sind High Performance Scanner dazu in der Lage, das gesamte IPv4 Internet in weniger als 45 Minuten nach Schwachstellen zu scannen. Da IPv6 eine weit höhere Anzahl an global eindeutigen Adressen zulässt, kann der alte Ansatz in der bekannten Form zur Analyse der IPv6 Infrastruktur nicht mehr verwendet werden. Diese Diplomarbeit beschäftigt sich mit der Adaption des bekannten IPv4 High Performance Scanners ZMap und der daraus resultierenden Fähigkeit IPv6 Scans durchzuführen. Desweiteren wird aufgezeigt, dass die große Anzahl an möglichen IPv6 Adressen alleine keinen Schutz vor Angriffen bietet. Es werden zwei verschiedene Vorgehensweisen untersucht, um mittels "High Performance ScanningÏnformationen über vorhandene IPv6 Infrastrukturen zu sammeln. Zusätzlich werden die Ergebnisse der durchgeführten Scans präsentiert und die dabei aufgetretenen Probleme analysiert.

Abstract

The rollout of the new Internet Protocol IPv6 requires an extensive transition phase which comes with a set of potential security vulnerabilities. For private individuals and companies as well as for the public sector, IT security plays an increasingly important role. Therefore it is important to point out the risks of the transition phase to be able to eliminate them in time. In the past, IP High Performance Scanning offered an efficient way to analyze the global Internet infrastructure and to identify potential vulnerabilities. IPv4 provided a manageable number of unique network addresses; this allowed High Performance Scanners to scan the entire IPv4 Internet in less than 45 minutes for vulnerabilities. Because IPv6 allows a much higher number of globally unique addresses, the old approach in the known form cannot be used to analyze the IPv6 infrastructure anymore. This thesis deals with the adaptation of the known IPv4 High Performance Scanner ZMap and its resulting ability to support IPv6. Furthermore it is shown that the large number of possible IPv6 addresses alone doesn not provide security. Two different approaches which could be used to gather information about existing IPv6 infrastructures by applying High Performance Scanning are discussed. In addition, the results of the performed scans are presented and the occurred challenges are discussed.

Contents

Kurzfassung **vii**

Abstract **ix**

Contents **xi**

1 Introduction **1**
1.1 Motivation and Problem Statement 1
1.2 Aim of the Work 2
1.3 Methodological Approach 3
1.4 Structure of the Work 3

2 Background and State of the Art **5**
2.1 Internet Protocol 5
2.2 Internet Control Message Protocol 15
2.3 Transition Phase 22
2.4 Network Penetration Testing 24
2.5 State-of-the-Art Network Scanners 24

3 Static and Dynamic Code Analysis of ZMap **27**
3.1 Setup Component 28
3.2 Sending Component 31
3.3 Receiving Component 32
3.4 Address Generation 34
3.5 Probe Module 35
3.6 Output Module 37
3.7 Summary and Conclusion 37

4 Adaptions of ZMap **39**
4.1 Setup Component 39
4.2 IPContainer Component 43
4.3 Sending Component 50
4.4 Receiving Component 51
4.5 Probe Module 52

4.6 Summary and Conclusion 53

5 Setup for Observations **55**
5.1 Finding Target Networks 55
5.2 Recognizing Patterns 59
5.3 Parameter Optimization of ZMapv6 60

6 Results **63**
6.1 Prefix Scanning 63
6.2 Address Pattern Scanning 69

7 Conclusion and Future Work **73**

A Addresses found through address pattern-based scanning **75**

List of Figures **79**

List of Tables **80**

List of Algorithms **81**

Acronyms **83**

Bibliography **85**

CHAPTER 1

Introduction

1.1 Motivation and Problem Statement

In 1998 IPv6 was specified as the new standard for addressing computers within the Internet [1] to solve problems of its predecessor IPv4 which came up in the years before. The most important problem of IPv4 is the limitation regarding the amount of available unique addresses [2]. IPv4 uses an address length of 32 bits and is able to allocate about four billion different addresses [1]. Nowadays, where many more households use devices that have to be connected to the Internet, IPv4 is not able to provide enough unique IP addresses anymore [2].

In the specification of its successor IPv6, the address length was extended to 128 bit [3]. This means an IPv6 address is four times longer than an IPv4 address and allows to address 3.4 x 10^{38} devices in total [4]; although the IPv4 address pool of the Internet Assigned Numbers Authority (IANA) ran out in 2011, roughly 0.2% Internet users used IPv6 at this time [2]. The reasons are twofold: First, the Internet Service Provider (ISP)s missed to adapt their infrastructure in time. Second, knowledge about the new protocol is not widespread. Missing knowledge and too little time for testing new implementations often lead to security holes and increase the risks of vulnerabilities in the IT infrastructure.

Lately, ISPs started to roll out their new infrastructure, also in Austria [5]. Transition technologies like Dual Stack (DS) and Dual Stack Lite (DS Lite) [6] enable their customers to stay connected with the IPv4 infrastructure and use the benefits of the new Internet Protocol simultaneously. Because of these technologies, users are not aware whether they are using IPv4 or IPv6 and whether they have been already connected to a potentially insecure and untested infrastructure; this fact allows ISPs to establish an IPv6 network without informing their customers.

This however might lead to distinct security risks. For example, the specification of IPv6 says [7], that every client gets its own unique global unicast address and is reachable by

any other participant of the Internet. Network Address Translation (NAT), a technology for IPv4 address saving, does not exist anymore with IPv6 and does not protect devices from unwanted access - although it was never intended to establish security anyway[3].

This makes it important to get insight on how far the IPv6 infrastructure is already distributed and what kind of security risks were produced by that. In IPv4 networks, Internet-wide scans are used to gain insight and gather statistical information about the distribution of commonly used technologies in the Internet. By sending probes to every client within the network, statements can be made about used protocols or versions of implemented products. Scanning is especially important for security: Responses that the sender receives are used to create answers to questions regarding existing vulnerabilities that could be exploited by attackers. In March 2014, when one of the most forceful flaws in OpenSSL, the Heartbleed vulnerability, was discovered, scans over the whole Internet were applied to get a general view on affected systems [8] and supported threat mitigation.

1.2 Aim of the Work

Due to the large amount of available IPv6 addresses, it is yet impossible to scan the entire IPv6 Internet economically [4], but the scanning approach is still promising to find answers to questions on today's IPv6 infrastructure in Austria. This effort is seen as an important step in supporting the transfer to a new technology as it is able to provide answers to certain questions on IPv6 deployment:

1. How far is IPv6 the deployment in Austria? Unsecured networks due to misconfiguration or ignorance are among the most dangerous situations occurring in the transition phase. Measuring the current extent of IPv6 deployment allows general statements of the ongoing transition phase. A list of deployed prefixes can be also used in future works regarding IPv6 research and vulnerability detection.

2. Is high performance scanning a valid approach to detect and uncover unknown IPv6 targets? Because of the large number of available IPv6 addresses, the problem of IPv6 scanning is to hit targets. This work should recommend a way to collect more responding addresses than the addresses that are publicly known e.g. by a Domain Name Service (DNS).

3. How well established is the routing infrastructure in Austria? A detailed analysis of IPv6 routes in Austria can expose single points of failure (SpoF), minor flaws and weak points regarding address planning and distribution. Furthermore a clever conditioning of the collected data will lead to a clear and plain overview of the existing router infrastructure.

1.3 Methodological Approach

The first step is researching the state-of-the-art of IPv6; this will include new technologies compared to IPv4 as well as the structure of the new protocol. That step is necessary to get thorough insight and knowledge about the new technology.

Due to scanning up to 2^{32} routed IPv6 subnets in a manageable amount of time, a high performance scanning approach promises success most likely. Therefore, the second step includes analyzing the source code of scanning tools like ZMap and MASSCAN. Both do not support IPv6 scanning out of the box, but the output of this analysis should highlight their different scanning strategies and should help implementing an IPv6 subnet scanner.

The third step contains the implementation of the IPv6 network scanner. Early research has shown that ZMap provides a solid base for implementation, but supporting IPv6 subnet scanning requires massive refactoring. This concerns mainly the following of parts:

1. The general handling of IPv6 addresses that contain a flexible control allowing switching between IPv4 and IPv6; detection of the global unicast address, the default route and gateway of the network; generation of IPv6 packets and matching outgoing with incoming packets.

2. The development of a special probe based on ICMPv6 (or an alternative protocol) including its dynamic generation. This contains a distinct implementation of the ICMPv6 protocol in C based on the definition of RFC4443 [9]. Every probe has to contain a unique ID that will be used to identify them.

3. Parsing and evaluating the incoming packets that were sent by target nodes. The packages that will be returned by answering nodes have to be parsed and the requested information has to be extracted, interpreted and correlated to the outgoing probes.

4. Development of an efficient permutation of target networks to ensure to not overload nodes. The permutation has to be complete and economical regarding computing power and memory.

5. The implementation of the IPv6 network scanner will be tested. The tests should investigate the advantages and disadvantages of IPv6 scanning in comparison to existing IPv4 scanning regarding performance and significance.

The fourth step deals with the discovery of the Austrian IPv6 Core Network. First, prefixes of well-known ISPs in Austria are scanned. This allows statements on IPv6 distribution and availability for end consumers. Second, an approach should be developed that allows to uncover unknown IPv6 addresses and to find new targets.

1.4 Structure of the Work

The remaining thesis is structured as follows:

Chapter 2 gives detailed information about the Internet Protocol version 4 and its successor Internet Protocol version 6. Also their related versions of ICMP are introduced.

Furthermore the terms Transition Phase, Network Penetration Testing and Network Scanners are discussed and examples are given.

Chapter 3 provides a detailed analysis of ZMap through performing a static analysis as well as a dynamic analysis of the program. The chapter highlights the most important code parts and shows how ZMap works.

Chapter 4 shows what kind of adaptions were required to make ZMap support IPv6. It gives information about what has been refactored and what had to be implemented completely new. Furthermore the chapter describes the algorithm that had to be developed to enable a flexible IPv6 address generation.

Chapter 5 lists the steps that where required to achieve new target addresses. It shows how to deal with the RIPE NCC database and what else can be done to collect many more addresses. Finally, it gives an introduction on how the execution parameters of ZMap should be chosen to perform meaningful scans.

Chapter 6 shows the final results of this master thesis and the results of the performed scans.

The last chapter gives a conclusion of the master thesis and recommendations for future works.

CHAPTER 2

Background and State of the Art

High performance scanning requires knowledge and understanding of the structures, characteristics and problems of network protocols. Therefore this chapter gives detailed information about the Internet Protocol version 4 and its successor Internet Protocol version 6. The behavior of both will be discussed and compared. Next the Internet Control Message protocol version 4 and version 6 will be introduced, by pointing out their most important tasks. They will be also compared to each other. Additionally the following paragraphs about the Transition Phase shows, what kind of technology exists, that enables to switch from IPv4 to the new protocol IPv6 smoothly. Next a short summary about Network Penetration Testing is given and after that, the term Network Scanner is discussed and some existing network scanners are introduced.

2.1 Internet Protocol

The Internet Protocol is a protocol which is used to enable logical addressing and communication of hosts within a network. The protocol is located at the Network Layer (Layer 3 of the Open Systems Interconnection Model (OSI model)) which is also called the Switching Layer. The communication in the Internet is packet oriented, therefore the route finding of packets is one of the most important tasks of the Network-Layer. It is carried out by the Internet Protocol. One routing node manages a set of networks and forwards the IP datagram to the next responsible node until it reaches its target [10].

Today there are two versions of the Internet Protocol available. They are discussed in this section.

2.1.1 IPv4

The Internet Protocol in version 4 was specified as standard Request for Comment (RFC) 791 in September 1981 [11]. The origin of this protocol was developed in the 70s by the

US Department of Defense [12]. This protocol is the basis of today's Internet. Although it has been stretched to its limits because of its finite number of unique addresses, IPv4 is still vital for the existing digital civilization.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td><td>31</td></tr>
<tr><td colspan="4">Version</td><td colspan="4">IHL</td><td colspan="8">Type of Service</td><td colspan="16">Total Length</td></tr>
<tr><td colspan="16">Identification</td><td colspan="3">Flags</td><td colspan="13">Fragment Offset</td></tr>
<tr><td colspan="8">Time to Live</td><td colspan="8">Protocol</td><td colspan="16">Header Checksum</td></tr>
<tr><td colspan="32">Source Address</td></tr>
<tr><td colspan="32">Destination Address</td></tr>
<tr><td colspan="24">Options</td><td colspan="8">Padding</td></tr>
</table>

Figure 2.1: IPv4 Header

The IPv4 header is illustrated in Figure 2.1 and according to RFC 791 the fields are specified as follows [11]:

- `Version`, 4 bits: The version field and the version of the protocol indicate the format of the header. If the field is set to 4 (like in this case), it can be assumed that the header is structured like the described fields.
- `IHL`, 4 bits: The Internet Header Length sets the length of the header in 32-bit words and indicates where the data section begins
- `Type of Service`, 8 bits: This field indicates the priority the service desires to be able to provide a specific quality level. It is for Quality of Service (QoS) realization.
- `Total Length`, 16 bits: The Total Length field specifies the length of the datagram including the header fields measured in octets.
- `Identification`, 16 bits: This ID is applied by the sender and helps to assemble fragments back to a complete datagram.
- `Flags`, 3 bits: The specified flags give information about the fragmentation of the datagram.
- `Fragment Offset`, 13 bits: The offset defines where the fragment belongs to in the datagram and is measured in multiples of 8 octets.
- `Time to Live`, 13 bits: To enable a network to discard undeliverable packets, the field Time to Live indicates the upper time the packet is allowed to remain in the network. The time is given in seconds but each module that processes the datagram has to decrease the field at least by one, even if it processes the datagram in less than one second. If the field is set to 0, the packet must be destroyed.

- `Protocol`, 8 bits: The content of this field is a number chosen from a list that contains various protocols. It specifies the protocol that follows after the Internet Protocol header. Thus it enables to assume which header can be expected in the payload of the Internet Protocol.

- `Header Checksum`, 16 bits: The Header Checksum assists by checking the integrity of the packet. This checksum is calculated by taking only the IP header into account. Because the content of the header field may change (i.e.: TTL), the checksum has to be calculated each time the packet was processed.

- `Source Address`, 32 bits: The IPv4 address of the sender.

- `Destination Address`, 32 bits: The IPv4 address of the receiver.

- `Options`, variable bits: Options are declared optionally, but every host and gateway must implement all options available. The transmission of the options is again optional. The field is used to transmit additional information that are required for instance for security purposes.

- `Padding`, variable bits: The padding field ensures that the header ends on a 32 bit boundary. The content of this field is not relevant.

An IPv4 address has a length of 32 bits. To improve the readability, an IPv4 address is segmented into four parts with a length of 8 bits (1 byte). This segments are separated by points and the numbers are printed in decimal format. This allows numbers between 0 and 255 for each segment, which means 256 different values per segment. In turn this 256 values in 4 segments allows to build 254^4 or 2^{32} unique addresses.

To enable gateways to route a packet over the world, an address is structured and divided into a network and a host part. The only thing each router the packets passes has to know is the next gateway to reach the target network of the packet. Only the last local network gateway has to be aware of the host addresses of its managed clients to deliver the packets correctly. This simplifies the routing table of each routing node which needs only one entry for each network and not for each host compared to Ethernet [13].

To group hosts to a network, three network classes named A, B and C were introduced (Table 2.1). Each class divides the 32 bits of an IPv4 address into a network part and host part of different lengths. Depending on the chosen class, a specific number of hosts are supported [14].

But this classification led to a non optimal address allocation, because networks were defined larger than required and thus addresses were wasted. The solutions was a concept of classification combined with a new notation, the Classless Inter-Domain Routing (CIDR) [15]. This concept does not follow the fixed classes for network lengths anymore. Instead the network is defined by bit-wise masking of the network part. This allows to define networks beyond the above-mentioned classes (Table 2.2) [12]. The notation

stipulates that the bits of the network part are written in decimal format after a slash that follows the IP address. For example: The network definition 192.168.30.0/23 indicates a network that contains host addresses starting with 192.168.30.1 and ending with 192.168.31.254. While it's a more comfortable way to write down IPv4 networks, it can also be seen as a short-term solution for the problem of scalability and maintainability.

Class	Network Bits	Host Bits	Total Networks	Total Addresses
A	8	24	127	16,777,216
B	16	16	16,384	65,536
C	24	8	2,097,152	256

Table 2.1: IPv4 Network Classes

Class	Class Mask	Classless Mask	Shortened
A	255.0.0.0	11111111.000000000.000000000.0000000000	/8
B	255.255.0.0	11111111.11111111.000000000.0000000000	/16
C	255.255.255.0	11111111.11111111.11111111.0000000000	/24
Classless	255.192.0.0	11111111.1100000000.0000000000.0000000000	/10
Classless	255.255.192.0	11111111.11111111.1100000000.0000000000	/18
Classless	255.255.240.0	11111111.11111111.1111000000.0000000000	/20

Table 2.2: IPv4 Classless Inter-Domain Routing

The concept IPv4 also stipulates a set of addresses which are not routed by any gateway. These so called private addresses are used to address local clients within a local network. Even if the old address classification is obsolete, it is still used to classify the private network segments. In each address class, a range is defined which is declared as private. Therefore it is not routed (Table 2.3). But thanks to CIDR, the classification alone gives no information about the network size. The number of hosts within a network is not restricted anymore by the address class.

Class	From	To
A	10.0.0.0	10.255.255.255
B	172.16.0.0	172.31.255.255
C	192.168.0.0	192.168.255.255

Table 2.3: IPv4 Private Addresses

To enable clients to establish connections to hosts of a global network anyway, NAT. This is also a short-term solution to manage the problem of scalability of the IPv4 concept. The concept stipulates one globally unique IPv4 address which is allocated to one interface of the local gateway. The client, that owns a private IPv4 address of a local network, sends the packets to its local gateway, if the target host is not part of the local network. The gateway replaces the source address of the packet by its own global IPv4

address and forwards the packet to the target or to the next gateway. The local gateway stores the information about the client and the target host. Then the response which belongs to this request will be edited again by the gateway before it will be forwarded to the host. The destination of the returning address will be replaced by the private host address of the client [16].

The application of NAT in IPv4 helps to save IPv4 addresses. Without NAT the Internet would have run out of globally unique IPv4 much earlier than it actually did. It would not be possible anymore to connect devices to the Internet and this would have thwarted the improvement in all technical sectors significantly. But the idea of NAT comes with a set of disadvantages. First, NAT forces the masking of the actual host. This side-effect is often misunderstood as a security feature, but it contradicts the original idea of IPv4 that every single member of the Internet gets its own globally unique IPv4 address. This would enable the identification of each host directly and over the globe. The second disadvantage is the loss of performance NAT comes with. The more connections are established over a NAT-ing gateway, the larger is the NAT-table that is required to store the connection information, which harms the data throughput [16].

This should point out, why NAT is just a short-term solution and why it especially important that a long-term solution like IPv6 was developed and should be provided globally in near future.

2.1.2 IPv6

IPv6 is the successor of the Internet Protocol version 4. The header of this new standard is specified in RFC 2460, which was defined in 1998 [7]. As its predecessor it is also located at the Network Layer of the OSI model and its main task is to define routes for packets. According to Stockebrand [2], the goal of IPv6 is twofold: First it should deal with all problems, that come with IPv4 (see subsections 2.1.1 and 2.1.3) that make IPv4 not efficient enough for today's use. Second, compared to IPv4, the Internet Protocol version 6 has been extended and comes with new features that where not required when IPv4 was specified.

Regarding RFC 2460, Figure 2.2 shows the IPv6 header. The following list describes its fields in further detail:

- `Version`, 4 bits: The version field and the version of the protocol indicate the format of the header. If the field is set to 6 (like in this case), it can be assumed, that the header is structured into the described fields.

- `Traffic Class`, 8 bits: This field is used to define a specific class the packet belongs to. It allows to carry out the concept of QoS. By selecting one class, the desired priority of this packet is defined as well.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Version | Traffic Class | Flow Level

Payload Length | Next Header | Hop Limit

Source Address

Destination Address

Figure 2.2: IPv6 Header

- `Flow Label`, 20 bits: A sender can choose a Flow Label to define packets that belong together. The label allows the routing node to handle these packets in particular. This is also required to realize the concept of QoS.
- `Payload Length`, 16 bits: The Payload Length field specifies the length of the datagram excluding the header fields, measured in octets.
- `Next Header`, 8 bits: The content of this field is a number chosen from a list that contains various protocols. It specifies the protocol that follows after the Internet Protocol header. Thus it enables the consumer to assume which header can be expected in the payload of the Internet Protocol.
- `Hop Limit`, 8 bits: Defines the number of hops a packet can take as maximum to finally reach its target. One hop is equal to one forward by a node. Each node that processes the packet has to decrements this counter. If the Hop Limit reaches 0, the packet has to be discarded. This enables to discard packets that cannot be delivered.
- `Source Address`, 128 bits: The IPv6 source address of the sender.
- `Destination Address`, 128 bits: The IPv6 source address of the receiver.

An IPv6 address has a length of 128 bits. The current addressing model and address representation is specified in RFC 4291 as followed [17]: An IPv6 address is divided into eight 16-bit pieces. The 16-bit segments are written in hexadecimal and are separated by a double point, e.g.: 2001:62a:4:430:43d:3cb5:fad1:79cd. In order to make writing IP addresses that contains long strings of zero bits easier, they can be replaced by "::". This

replacement may only appear once in an address. It allows to represent e.g. the IPv6 loopback address "0:0:0:0:0:0:0:1" as "::1" and may also shorten a unicast address from "2001:0:0:430:0:0:0:79cd" to "2001:0:0:430::79cd".

According to RFC 4291 there are three types of addresses [17]:

1. Unicast: An unicast address identifies a single interface of a node. This means that a packet that is sent to an unicast address is delivered to the interface that is identified by that address.
2. Anycast: An anycast address identifies a set of interfaces, which typically belong to different nodes. A packet that is sent to an anycast address is delivered to one interface that is specified in the set of interfaces. The interface is usually chosen by the least distance that is calculated by the routing protocol.
3. Multicast: A multicast address identifies a set of interfaces, which typically belong to different nodes. A packet that is sent to a multicast address is delivered to all interfaces that are specified in the set of interfaces.

Prefixes of IPv6 addresses that declare a network and allow subnetting are also written in CIDR notation [17]: <ipv6-address>/<prefix-length> where <ipv6-address> represents the IPv6 address written in hexadecimal and <prefix-length> is written in decimal and defines the leftmost contiguous bits that represent the prefix and with it the network ID.

RFC 4291 also contains a classification of IPv6 addresses which classifies the IPv6 address space on the basis of the prefix [17]: Table 2.4.

Address Type	Binary Prefix	CIDR Notation	Detailed specification in
Unspecified	00...0 (128 bits)	::/128	RFC 4291[1]
Loopback	00...1 (128 bits)	::1/128	RFC 4291[2]
Multicast	11111111	FF00::/8	RFC 7346[3]
Link-Local Unicast	1111111010	FE80::/10	RFC 4193[4]
Global Unicast	(everything else)		RFC 3587[5]

Table 2.4: IPv6 Address Space Classification

The Link-Local Unicast and the Global Unicast addresses are required as background for this master thesis, therefore they are described in more detail in the following paragraphs:

Link-Local Unicast
According to RFC 4193 Link-Local Unicast addresses are specified as the "private" IPv6

[1] https://tools.ietf.org/html/rfc4291
[2] https://tools.ietf.org/html/rfc4291
[3] https://tools.ietf.org/html/rfc7346
[4] https://tools.ietf.org/html/rfc4193
[5] https://tools.ietf.org/html/rfc3587

addresses of an interface of a node. They are not routed globally by any gateway but may be routed inside a limited area e.g. sites. The characteristics of a local IPv6 unicast address is defined as follows [18]:

- The prefix of Link-Local Unicast address is globally unique with a high probability.
- The prefix is well-known to enable easier filtering on site borders.
- Connecting sites that using Link-Local Unicast addresses for their clients can be combined or privately interconnected without being afraid of address conflicts e.g. through inter-site Virtual Private Network (VPN).
- The Link-Local Unicast address is ISP independent. That allows communication within a site without having any permanent or intermittent Internet connectivity.
- It can be assumed, that it will not come to an address conflict if a Link-Local Unicast address is accidentally leaked outside of a site via routing or DNS.
- Applications do not have to differentiate between addresses of a local or a global scope.

Global Unicast Address
The IPv6 Global Unicast Address is the IPv6 equivalent of an IPv4 public address. Usually an address of this class is globally reachable via the global routing infrastructure [13]. According to RFC 3587 Table 2.5 shows the general format of a global unicast address [19].

n bits	m bits	128-n-m bits
global routing prefix	subnet ID	interface ID

Table 2.5: General Format of an IPv6 Global Unicast Address

The value of the global routing prefix follows a hierarchical structure, is managed by Regional Internet Registry (RIR) and ISPs and is assigned to a site. The subnet ID allows to create subnetworks within a site. It is administrated by the site administrator. The Interface ID defines the part that is associated with the hardware interface the address is assigned to [19].

From the large address space that is reserved for global unicast addresses (Table 2.4), IANA assigns only addresses under the prefix 2000::/3 at the moment. RFC 3177 recommends the following rules [20]:

- The general size of a prefix that is allocated to a site should be /48.
- If only one subnet is required, than the prefix should be specified with /64. This results also in an interface ID length of 64 bits.

- If it is absolutely known that only one device is connected, the prefix can be defined with /128.

- Small and large enterprises and home network subscribers should receive a /48 prefix.

- large subscribers should receive a prefix with a length of 47 bits, or multiple /48 prefixes

- Mobile networks should be specified as a /64 prefix.

Table 2.6 takes those recommendations into account, adds the information about the prefix assignment of IANA and shows a more specific structure of a global unicast address [19].

3 bits	45 bits	16 bits	64 bits
001	global routing prefix	subnet ID	interface ID

Table 2.6: Specific Format of an IPv6 Global Unicast Address

But searching the Réseaux IP Européens Network Coordination Centre (RIPE NCC) database (see section 5.1) shows, that the recommendations of RFC 3177 are not observed. Also in Austria, several companies (e.g. Zumtobel Group AG, Red Bull Media House GmbH) have registered a /32 global routing prefix.

The Regional Internet Registries (RIRs) reconsidered these recommendations in 2005, which made the described rules in RFC 3177 obsolete. In reaction to this, IETF reviewed the actually implemented and provided address architecture and created a new RFC (RFC 6177) which provides a best practice guide for end site assignments. It clarifies, that allocating mainly 48 bit prefixes does not provide the required flexibility for the broad range of all end site assignments and is therefore not recommended anymore. The operational community has to make decisions about the size of the assigned address space, depending on the local requirements. It softens the rules of RFC 3177 indeed, but also clarifies, that scalability and growth over long time periods must be ensured [20][21].

2.1.3 Comparison of IPv4 and IPv6

Table 2.7 gives a brief comparison between IPv4 and its successor IPv6.

Feature	IPv4	IPv6
Address length	An IPv4 address has a length of 32 bits. This allows to build 2^{32} globally unique addresses [11].	An IPv6 address has a length of 128 bits. This allows to build 2^{128} globally unique addresses [7].
Address format	An IPv4 address is defined by four 8-bit segments, which are written in decimal and separated by points [11].	An IPv6 address is defined by eight 16-bit segments, which are written in hexadecimal and separated by double-points [7].
Address configuration	There are three ways to configure an IPv4 address: Manually; with the help of Dynamic Host Configuration Protocol (DHCP) [22]; by using dynamic configuration of link-local addresses [23]	There are three ways to configure an IPv6 address: Manually; with the help of Dynamic Host Configuration Protocol, version 6 (DHCPv6) [24]; by using IPv6 Stateless Address Autoconfiguration [25]
Header length	The header size of an IPv4 datagram is variable. Padding bits have to be used to ensure a boundary of 32 bits[11]	The header length of an IPv6 datagram is fixed. IPv6 offers a set of extension headers, that enable to define the options which are similar to the option field of the IPv4 header [7].
Integrity Check	The IPv4 header provides an integrity check by calculating a checksum. This checksum has to be calculated by every node that processes the packet [11].	There is no integrity check on Network-Layer in the concept of IPv6. Compared to IPv4 this relieves the processing nodes and boosts transmission performance because they do not have to calculate new checksums after processing each packet [26].
End-to-end connectivity	To save addresses, NAT was developed. It allows to hide a set of hosts behind one public IP address. But the concept of NAT violates the original idea of end-to-end connectivity [16].	The large number of globally unique IPv6 addresses allows to fulfill the concept of end-to-end connectivity without translation mechanisms in between. Nevertheless NAT is also specified for IPv6 but is yet in experimental status [27].
Security	When IPv4 was developed, the security perspective was given only a minor priority. Therefore all security features have to be provided by the application [28].	IPv6 specification comes with a set of Network-Layer security features like encryption and authentication of communications [28].

Table 2.7: Comparison between IPv4 and IPv6

2.2 Internet Control Message Protocol

The Internet Control Message Protocol is a well-known protocol that comes in combination with the Internet Protocol. It uses basic support of the Internet Protocol, like a higher level protocol, but in fact it is an inherent part of the Internet Protocol. With the introduction of IPv6 also a new version of ICMP, ICMPv6 has been developed. While ICMPv4 was mostly used for diagnostic purposes, ICMPv6 has much more functional responsibility in the concept of IPv6. Both protocols are discussed and compared in the following subsection.

2.2.1 ICMPv4

The Internet Control Message Protocol version 4 is specified in the RFC 792 which was written in 1981. The idea of this specification is a control mechanism that gives feedback for problems in the communication environment. But ICMPv4 does not improve the reliability of IPv4. If reliability is required, it has to be considered in the implementation of the higher level protocols. Usually ICMPv4 reports errors that can occur during the processing of packets. ICMPv4 messages are using the basic IPv4 header, where the first field of the ICMPv4 header gives information about the message type. Depending on this type, the ICMPv4 header is structured differently [29].

According to RFC 792, ICMPv4 supports 11 different message types [29]:

- Type 0: Echo Reply
- Type 3: Destination Unreachable
- Type 4: Source Quench
- Type 5: Redirect
- Type 8: Echo
- Type 11: Time Exceeded
- Type 12: Parameter Problem
- Type 13: Timestamp
- Type 14: Timestamp Reply
- Type 15: Information Request
- Type 16: Information Reply

The most relevant message types for this master thesis are: Echo Request, Echo Reply, Destination Unreachable and Time Exceeded. They are described in more detail in the following subsections:

Echo Request and Echo Reply Messages: This pair of message types is often used to check if a target is reachable within a network. The most famous tool that uses an implementation of these two message pairs is "Ping" [6] [30]. Figure 2.3 shows the header structure of both messages. The source address of the IPv4 header of an Echo Request message is the destination address of the IPv4 header of an Echo Reply message and vice versa. The code is 0 for both messages. ICMPv4 includes an integrity check by calculating a checksum. This is done by building the 16-bit ones' complement of the ones' complement sum of the ICMP message starting with the ICMP Type. The identifier and sequence fields help to match Echo Request messages to corresponding Echo Reply messages. The payload of an Echo Reply message has to contain the complete Echo Request message, that triggered the Echo Reply message [29].

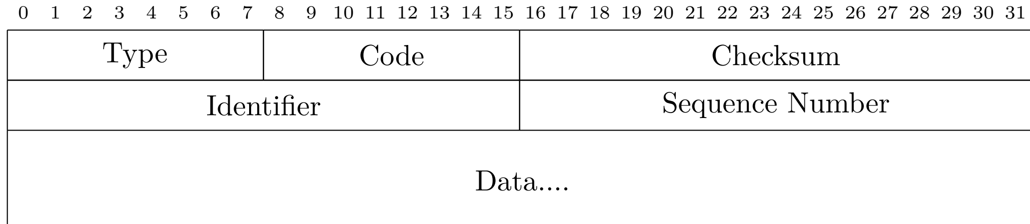


Figure 2.3: ICMPv4 Header Echo Request/Reply

Destination Unreachable: This message type provides a number of different codes. Summarized, this message will be sent, if the gateway is not able to deliver the packet according to its routing table, if the the target host does not provide an active service on the target port, or if the gateway has to fragment the packet but is not allowed to because the "Do not Fragment" flag is set. Figure 2.4 shows the ICMPv4 header of such message type [29].

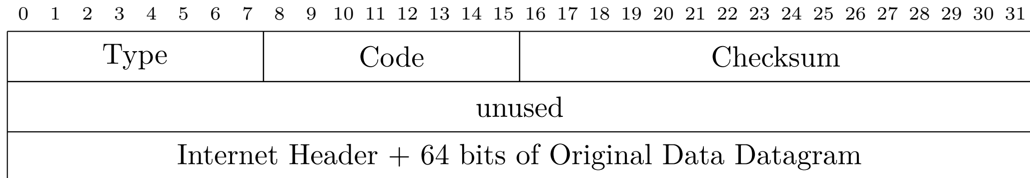


Figure 2.4: ICMPv4 Header Destination Unreachable

The following message codes are supported by ICMPv4 messages of type "Destination Unreachable" [29]:

- Code 0: network unreachable.
- Code 1: host unreachable.

[6] `http://linux.die.net/man/8/ping`

- Code 2: protocol unreachable.
- Code 3: port unreachable.
- Code 4: fragmentation needed and DF set.
- Code 5: source route failed.

Time Exceeded: The ICMPv4 header structure for this message type is the same as for the type "Destination Unreachable" (Figure 2.4, but it provides two different codes: 0 stands for "time to live exceeded in transit" and 1 stands for "fragment reassembly time exceeded". Each gateway that receives a packet has to check the IPv4 header field Time to Live (TTL). Whenever the packet is processed by a gateway this field has to be decremented. If the field reaches the value 0, the packet has to be discarded and an ICMPv4 message of type 11 has to be sent to the source address of the discarded packet. The same message, but with the second code value, has to be sent if a fragmented datagram cannot be reassembled within a specific time because of missing fragments [29].

2.2.2 ICMPv6

The Internet Control Message Protocol, version 6 was specified in 2006 and is documented in RFC 4443. After standardizing IPv6, it was decided that ICMP will be again the protocol to support and extend the functionality of the next-generation Internet Protocol [9]. Compared to ICMPv4, the main tasks of ICMPv6 are again forwarding information and reporting errors, but enhancements made ICMPv6 to a much more important protocol. Blocking all ICMPv6 messages could lead to an inoperable IPv6 [9][28].

According to RFC 4443 the ICMPv6 header (Figure 2.5) is announced by setting the value of the Next Header field of the IPv6 header to the value 58. The header contains three fields. The type indicates the subsequent structure of the header. The code field groups the message types into smaller subcategories. The checksum helps to detect data corruption. It is calculated by summing up the entire ICMPv6 message including a "pseudo-header" of IPv6 header fields and building the 16-bit ones' complement of the ones' complement sum [9].

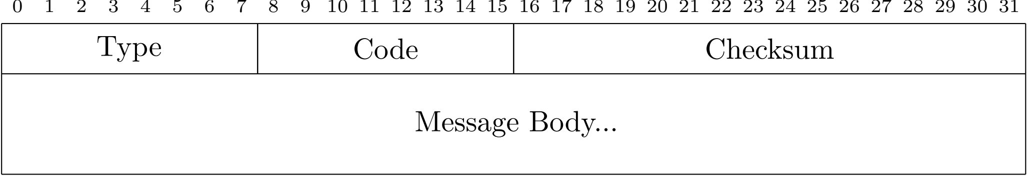


Figure 2.5: ICMPv6 Message Format

RFC 4443 defines only two categories of ICMPv6 message types: ICMPv6 error messages and ICMPv6 informational messages. This specification includes 13 type values [9] which

are quite similar to the message types of ICMPv4. But by implementing extension of the ICMPv6, more message types were required. Because the different extensions were specified in own RFCs, IANA provides a list of all message types available. At the moment this list[7] contains 43 entries. This subsection discusses the message types which are especially important for IPv6 operations and this master thesis. The most important types are Echo Request, Echo Reply, Destination Unreachable, Time Exceeded and the message types that belong to neighbor discovery namely Neighbor Solicitation, Neighbor Advertisement, Router Solicitation and Router Advertisement.

Echo Request and Echo Reply Messages: Like ICMPv4, ICMPv6 provides messages for testing the reachability of hosts. Also the header looks exactly the same (see Figure 2.6). Echo Requests (type = 128 and code = 0) are often used for network scanning approaches. The Echo Response (type = 129 and code = 0) also includes the invoking message in the payload. An IPv6 compatible version of the tool ping (parameter -6) or ping6[8] allows to send Echo Request messages to IPv6 destination addresses [9].

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Bits 0–7	Bits 8–15	Bits 16–31
Type	Code	Checksum
Identifier		Sequence Number
Data....		

Figure 2.6: ICMPv6 Header Echo Request/Reply

Destination Unreachable According to RFC 4443, this type of message should be generated if the invoking packet cannot be delivered. However, it must not be generated if the reason of not delivering is congestion. The payload of the packet has to contain as much of the invoking packet as possible and the destination address is its sender address. The value of the type field of the header (see Figure 2.7) is 1 and there are several message codes available which give information about the reason of rejection [9]:

- Code 0: No route to destination
- Code 1: Communication with destination administratively prohibited
- Code 2: Beyond scope of source address
- Code 3: Address unreachable
- Code 4: Port unreachable

[7] `https://www.iana.org/assignments/icmpv6-parameters/icmpv6-parameters.xhtml`

[8] `http://linux.die.net/man/8/ping6`

- Code 5: Source address failed ingress/egress policy
- Code 6: Reject route to destination

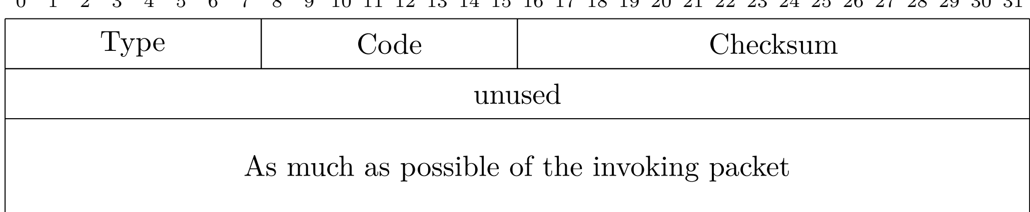


Figure 2.7: ICMPv6 Header Destination Unreachable

Time Exceeded: Similar to the time to live field (TTL) of the IPv4 header, the IPv6 header contains a field called hop limit (see Figure: 2.2). The value of the field has to be decremented whenever it is processed by a node [7]. If the value reaches 0, the packet has to be discarded and an ICMPv6 message should be sent to the sending host. Reaching the value 0 can be an indicator for a routing loop or a too low chosen initial value for that field [9].

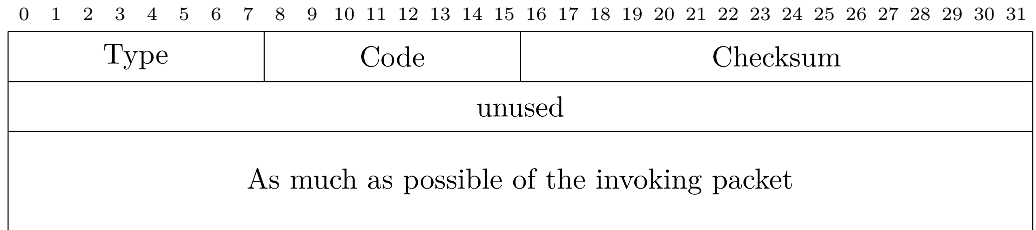


Figure 2.8: ICMPv6 Header Time Exceeded

Neighbor Discovery for IP version 6

The implementation of neighbor discovery for IPv6 is completely carried out by extensions of ICMPv6 that are specified in the RFC 4861. Neighbor discovery is one of the reasons why ICMPv6 is important for basic IPv6 functionality and why blocking the complete ICMPv6 traffic would also block features of IPv6 [9][28]. In comparison to IPv4, the IPv6 neighbor discovery corresponds to a combination of the IPv4 Address Resolution Protocol (ARP), ICMPv4 Router Discovery and ICMPv4 Redirect. It also supports mechanisms to Neighbor Unreachability Detection which was never specified for IPv4 [31]. The most important message types regarding neighbor discovery are discussed in the following passages.

Neighbor Solicitation and Neighbor Advertisement The goal of Neighbor Solicitation (NS) (Type = 135)[31] and Neighbor Advertisement (NA) (Type = 136)[31]

message types is the substitution ARP in IPv6 [2]. A NS message is sent by a node, if it requests a layer-link address of another node. Next to the already known fields of the ICMPv6 header, the NS header contains a target IPv6 address (see Figure 2.9). The target address is a multicast address, if the requesting node needs to resolve an IPv6 address. The target address has to be a unicast address, if the requesting node has to check the reachability of a target node. When the nodes sends NS messages, the node also provides its own link-layer address. This is done by using the `Options` field of the header (see Figure 2.9) [31].

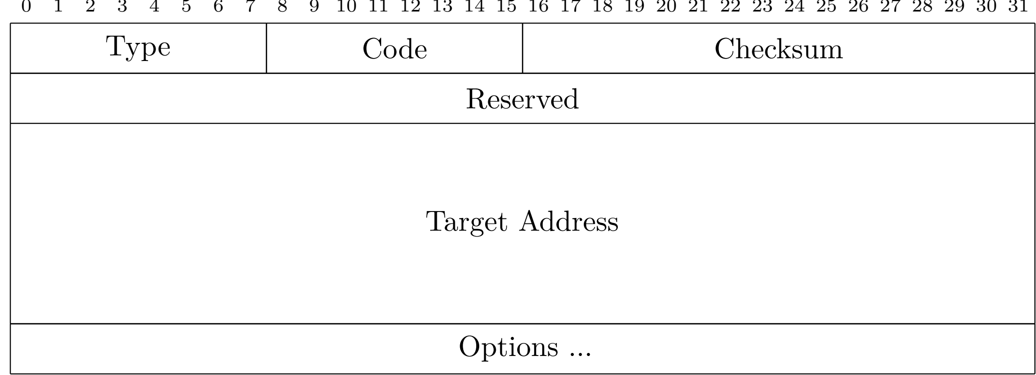


Figure 2.9: ICMPv6 Header Neighbor Solicitation

As response to a NS message, the target node sends a NA message to the invoking node. In contrast to the NS messages, the `Target Address` field (see Figure 2.10) of a NA message must not contain a multicast address. If the advertisement is solicited, the field is filled with the the value of the `Target Address` field of the corresponding NS message. If the NA message is sent without any prompt, the field contains the IPv6 address of the node, whose link-layer address has changed [31].

The ICMPv6 header of NA message also provides a set of flags which allows to give information about the message in further detail [31]:

- `R`:The router flag is set, if the sending node is a router. The flag is used by the Neighbor Unreachability Detection mechanism to detect a router that has changed to a host node.
- `S`:The solicited flag is set if the NA message is sent as response of a solicitation. It is used as a reachability confirmation for the Neighbor Unreachability Detection.
- `O`: The override flag tells the target node that the link-layer address of the sending node has changed and the local Neighbor Cache entry of the target node has to be updated.

Router Solicitation and Router Advertisement The Router Solicitation (RS) and Router Advertisement (RA) message types are required for the Stateless Address Autocon-

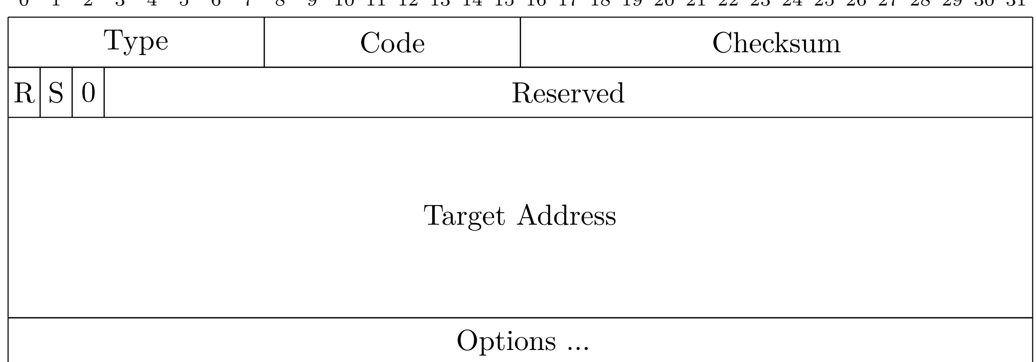


Figure 2.10: ICMPv6 Header Neighbor Advertisement

figuration (SLAAC) feature of IPv6. In IPv6 a task of routers is to manage the network prefixes that are assigned to network segments. This comes with a set of advantages compared to the DHCP solution for address distribution. If a host needs an IPv6 address, it asks all routers connected to the network for additional prefixes by sending RS messages. All routers reply with RA messages that contain a set of prefixes and the information, if the corresponding router provides routing services. For each prefix the host receives, it creates an address for the interface and checks if another host within this network already uses this address by performing the Duplicate Address Detection algorithm [25][31][2].

The ICMPv6 header of RS messages (see Figure 2.11) contains the value 133 in message type field and 0 in the code field. The options field contains the link-layer address of the source hosts if the link-layer address is known [31].

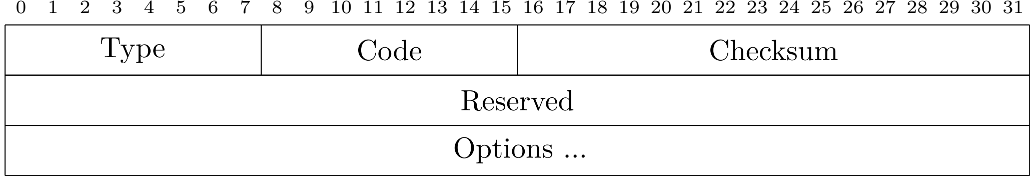


Figure 2.11: ICMPv6 Header Router Solicitation

The RA response of the router has the message type id 134 and also the code 0. This message type can be also sent periodically by the router without any solicitation. The header (see Figure 2.12) contains a value for the hop limit (field `Cur Hop Limit`), the host should put into the hop limit field of the IPv6 header of its packets. The two 1-bit fields `M` and `O` tell the requesting host if a address is available via DHCP (the M-flag is set) or via other configuration (the O-flag is set). The `Router Lifetime` specifies in units of seconds the time a router is defined as default router. The field `Reachable Time` lets the host assume how many milliseconds the neighbor node is reachable. This field is used for the Neighbor Reachability Detection algorithm. The router is able to deliver

the source link-layer address of the interface the message was sent from, the Maximum Transmission Unit (MTU) of the link and the prefix information via the `Options` block [31].

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td><td>31</td></tr>
<tr><td colspan="8">Type</td><td colspan="8">Code</td><td colspan="16">Checksum</td></tr>
<tr><td colspan="8">Cur Hop Limit</td><td>M</td><td>O</td><td colspan="6">Reserved</td><td colspan="16">Router Lifetime</td></tr>
<tr><td colspan="32">Reachable Time</td></tr>
<tr><td colspan="32">Retrans Timer</td></tr>
<tr><td colspan="32">Options ...</td></tr>
</table>

Figure 2.12: ICMPv6 Header Router Advertisement

2.2.3 Comparison of ICMPv4 and ICMPv6

After the specification of IPv6 was done, it was clear, that the new Internet Protocol needed an own implementation of the Internet Message Control Protocol. The main difference between ICMPv4 and ICMPv6 therefore obviously is the dependency of the related version of the network protocol. This means ICMPv4 only works together with IPv4 and ICMPv6 needs IPv6 as underlying protocol. ICMPv4 requires the value 1 in the `Protocol` field of the IPv4 header [29]; for ICMPv6 the value 58 is expected in the IPv6 header field `Next Header` [9].

The standardization of IPv6 and ICMPv6 tried to improve the efficiency by avoiding mistakes that where made in IPv4 and ICMPv4 specification because of not being long-sighted. All message types of ICMPv6 provide segments in their headers that are not yet specified and which are reserved for future extensions. This improves flexibility and longevity of ICMPv6 compared to ICMPv4.

ICMPv4 is important for IPv4 diagnostics and analysis, but ICMPv6 is much more important for a lot of IPv6-unique features. The new protocol is responsible for a set of mechanisms, that were carried out by separate protocols in IPv4. Especially the Neighbor Discovery Protocol, which uses ICMPv6, provides a multitude of improvements and allows for instance IPv6 address configuration or duplicate address detection without the need of a third transport protocol [31].

2.3 Transition Phase

Because IPv4 is the most implemented and used network protocol in the world [32], it is not possible to define a day zero on which all Internet connections have to be established over IPv6. Rather a transition phase that allows the coexistence of IPv6 and IPv4 was planned. To realize this, several technologies that enable hosts to use IPv6 and IPv4

simultaneously were introduced. The most common technologies are discussed in this section. Having knowledge about them can be seen as a precondition to be capable of creating setups for high performance scans and recognizing correlations between allocated IPv6 addresses. Furthermore, the transition phase comes with multiple security risks. Misconfiguration, missing knowledge and high complexity can lead to distinct security holes if IPv6 is provided parallel to IPv4 within a network [33] [32] [34].

2.3.1 Dual Stack

Dual Stack, also called Dual IP Layer [34], is a technology that allows to equip the host and the components of a network infrastructure with both Internet Protocols. This enables the host to be compatible to both connection methodologies and to distinct between IPv6 and IPv4 packets on one or more links [34][32].

A modification of Dual Stack is the related technology Dual Stack Light, which is often used by large ISPs. The basic concept is to save public IPv4 addresses by defining only the IPv6 network as globally routable. The IPv4 address, which is deployed to the host, is part of a private IPv4 network. The IPv4 packets are encapsulated over the IPv6 traffic and are sent to an Address Family Transition Router (AFTR). The tasks of the AFTR is to use the NAT technology to translate the private IPv4 address to a public IPv4 address and establish the connection. The response is again translated, encapsuled and sent back to the host via the IPv6 infrastructure [6]. Usually one AFTR is responsible for multiple ISP customers. The ISP saves IPv4 addresses by sharing one public IPv4 between them.

Next to managing the shared IPv4 address securely and sealing the traffic of the sharing customers [6], another security risk is ignorance of consumers. Dual Stack light allows ISPs to deploy the IPv6 infrastructures directly to their customers without informing them. Because the IPv6 addresses are globally reachable, attackers could access the end nodes if appropriate security measures like adaption of the gateway firewall configuration are not made.

2.3.2 Tunneling

A tunnel mechanism allows to enter an IPv6 network from a IPv4-only host by encapsulating the IPv6 traffic over IPv4 which means that the whole IPv6 packet is transported in the payload of an IPv4 packet [32]. The encapsulation is the task of the tunnel entry point. The packet is then transported to the exit node, where it has to be decapsulated and routed to its target address [34]. There are several tunnel techniques available. The most common ones are 6to4, ISATAP and Teredo [35][36][37]. Each of them come with multiple security risks [33].

2.3.3 IPv4/IPv6 Translation

RFC 6144 describes a framework for IPv4/IPv6 translation, which is also seen as a medium-term transition strategy. The document focuses an all kinds of situations where

translation is required. Summarized it describes a set of systems that communicate using IPv4 only and their interoperability with systems that communicate using IPv6 only. The RFC breaks it down into specific translation classes that are discussed in details. From a technical point of view, the described IPv4/IPv6 translator extends the existing Stateless IP/ICMP Translator Algorithm (SIIT), which translates IPv4 and IPv6 packet headers of stateless connections, by a stateful translation [38].

2.4 Network Penetration Testing

Network penetration testing is a kind of penetration testing that focuses on exposing weaknesses in the network infrastructure of a company. Other types of penetration tests that focus on other threats within a company are: Application Penetration Tests, Periodic Network Vulnerability Assessment and Physical Security Penetration Tests [39].

Regarding Andrew Whitaker and Daniel P. Newman, the following types of penetration tests can be distinguished [40]:

- `Black-box test`: This type of test simulates an attack from outside the company. A penetration test is called a black-box test, if the tester has no special knowledge about the network or the infrastructure.
- `White-box test`: The tester, who performs a white-box test, owns full knowledge about the infrastructure of the company. Usually, he has access to the documentation of the network components. This tests simulates the worst case scenario but is most accurate.
- `Gray-box or crystal-box test`: The tester owns partial knowledge about the company. It simulates an employee that tries to harm the company by attacking the infrastructure.

2.5 State-of-the-Art Network Scanners

Usually a network penetration tester uses a set of tools that help to gather information about their targets and to discover the infrastructure [41]. Network scanners are one of these tools. The goal of a network scanner is to detect and search for hosts within a target network. This is done by sending requests of a specific protocol type to a randomly chosen IP address within an address pool. By analyzing the packet the target replies, statements can be made about the host. Several network scanners also provide a port scanning function that allows to scan a specific range of ports of a target host. Fingerprinting or banner grabbing allow to gain more detailed information about the discovered host [42] [8].

This master thesis distinguishes between normal network scanners and high performance network scanners. Usually a normal network scanner comes with additional functionality

that allows to grab as much information as possible from a target host. The disadvantage of this kind of scanner is the low speed. Because the focus is set on gathering information in a lower IP address range and analyzing the state of the connections to a target host, it is rather not useful for scanning the entire IPv6 Internet. In contrast, high performance scanner burst out as much packets as possible and analyze only the responses they receive. They do not care about connection states and also do not have to remember the hosts they tried to reach. In the following sections, commonly known network scanners of both types are discussed.

2.5.1 ZMap

ZMap is a high performance scanner, is written in C and was developed by the University of Michigan. It is able to survey the entire IPv4 address space in under 45 minutes from user space on a single machine with no special adaptions [4] and under five minutes when the kernel driver `PF_Ring` is loaded [43]. ZMap is an open-source project and allows multiple options to provide flexibility and protection of risky side effects. Flexibility is required to create different test setups that will fit into the special needs of the researcher without changing the core code of the tool. A fast network scanner like ZMap also has to ensure that a burst of small packages will not end up in a Denial-of-Service (DoS) attack of target devices located in a scanned network segment. Currently, ZMap does not support IPv6 scanning out of the box. This is because the amount of IP addresses is too large for scanning by brute forcing address by address. As the developers of ZMap say in their paper, for scanning the entire IPv6 address space, new technologies and methodologies have to be researched [4].

In April 2016, the institute of Informatics of the Technical University of Munich published a paper about IPv6 Scanning towards a comprehensive hitlist. The goal of this work was to measure the availability of known IPv6 hosts over time. They used the ZMap source code and implemented basic IPv6 functionality. With this extension, ZMap is able to accept a list of IPv6 addresses as input and send packets to the related IPv6 hosts. This is done periodically. The result is a overview of how long a host is reachable over an explicit IPv6 address grouped by the network protocol and network port. Furthermore, statements can be made about the routed and used IPv6 prefixes [44].

2.5.2 MASSSCAN

MASSCAN also is a free high performance scanner. It's also written in C. According to its documentation it is able to scan the entire IPv4 Internet in under six minutes by using the special driver `PF_Ring` [45]. Like ZMap, MASSCAN includes randomization of the target IP addresses to ensure that the network infrastructure will not get overloaded; but Durumeric proved [43], that ZMap is much better in creating addresses randomly. MASSCAN is actually not able to scan ranges of the IPv6 network either.

2.5.3 NMAP

A famous and widely used network scanner is Nmap ("Network Mapper"). Nmap is currently available for download at version 6. It supports basic IPv6 functionality since 2002 and since version 6 also frequently used features like OS detection, advanced host discovery and raw-packet IPv6 port scanning [42]. It is not capable of scanning IPv6 address ranges and uses alternative ways of network reconnaissance. Nmap is not a high performance scanner. The developer of ZMap demonstrated [4], that ZMap is much faster than Nmap without losing accuracy because it is especially designed for high performance scans in IPv4 networks. Nmap instead provides more optional features. Therefore ZMap offers a good basis for scanning IPv6 addresses in respective to speed and required time.

CHAPTER 3

Static and Dynamic Code Analysis of ZMap

A static code analysis is a software testing methodology that works by analyzing the source code of software without execution. It is a white box software testing approach, which means that the complete source code of the program is available for analysis. Liggesmeyer brings up the following characteristics [46]:

- An execution of the program is not absolutely necessary.
- Every static analysis can be done without the support of a computer.
- There will not be any special test cases chosen.
- It is not possible to make complete statements about correctness and reliability of the analyzed program.

Furthermore, Liggesmeyer describes the execution of a program as the simplest form of dynamic software tests and analysis [46]. Different user inputs and varying input parameters lead to distinct behavior of the program. As additional support during the dynamic analysis, one of most popular and widely spread debuggers, GNU Debugger (GDB), provided by Free Software Foundation [47], can be used to debug the program step by step. This approach makes it easier to understand the software's behavior.

In the context of this master thesis both approaches had to be used to gain knowledge about the way ZMap works. The goal was to determine the main components of the program and to find out which and how many parts have to be adapted to extend ZMap by a new feature; namely the scanning of IPv6 networks. The source code of ZMap is

available for download from Github.com[1] and can be used under the Apache open source license[2].

A more general view on the architecture of ZMap gives the paper "Fast Internet-wide Scanning and Its Security Applications" written by the developers of ZMap [4]. Next to detailed information about the scanner and some use cases, there is also a figure that shows the architecture of ZMap and how the single parts fit together, see Figure 3.1.

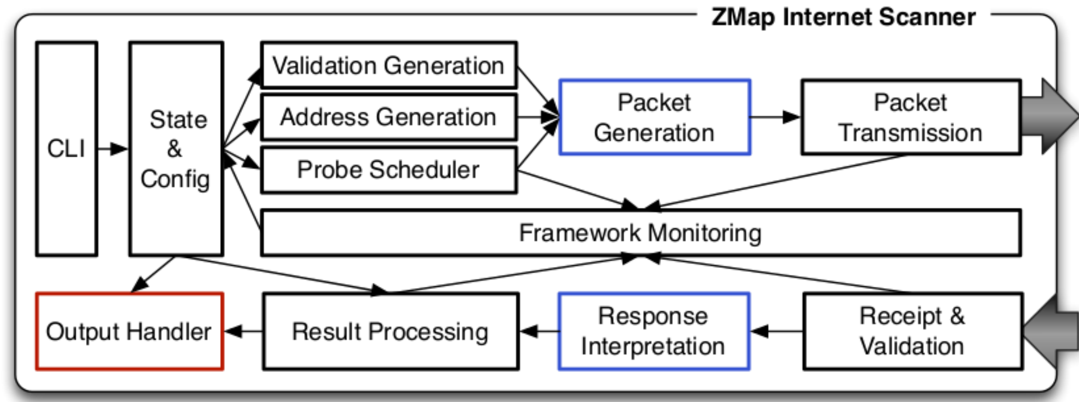


Figure 3.1: ZMap architecture [4]

The outcome of the first static code analysis has however shown, that this graphic (figure 3.1) is rather an abstract perspective on ZMap. To understand the real code components of the program better, Figure 3.2 provides a more technical perspective. It divides the code into the *Setup Component*, the *Sending Component*, the *Receiving Component*, the *Address Generation*, the *Probe Module* and the *Output Module*. The following subsections describe the main functions of these components.

3.1 Setup Component

The first tasks of the Setup Component is parsing all input parameters that were given by the user through the Command-line Interface (CLI). It stores them in the matching variable of the struct `state_conf`. After that, all given parameters can be accessed from any point of the program. All input parameters that are supported by ZMap are auto generated by GNU Gengetopt in Version 2.22.6 [3]. GNU Gengetopt is a tool to generate C code that parses the command line arguments `argc` and `argv`, which are available in every C and C++ program. The tool is free to use and can be modified under the license of GNU General Public License [48].

[1] https://github.com/zmap/zmap
[2] https://www.apache.org/licenses/LICENSE-2.0
[3] https://www.gnu.org/software/gengetopt/gengetopt.html

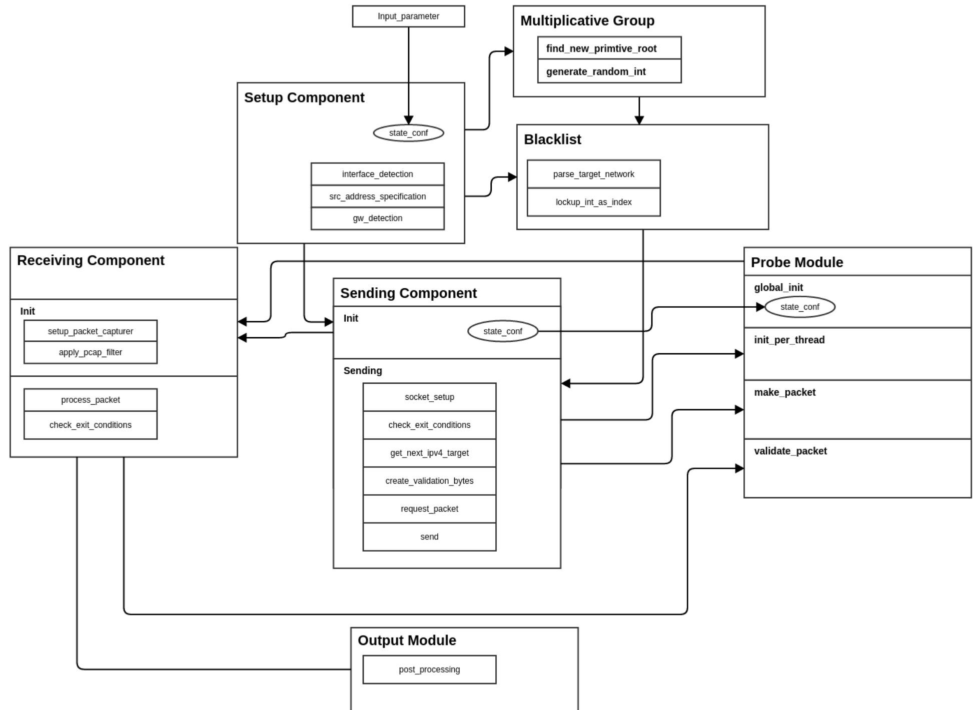


Figure 3.2: ZMap dependencies from technical point of view

The second task is to evaluate the given execution parameters and to compute missing parameters that are relevant for the setup of the sending and receiving processes. The most important parts of this task are:

- Determining the interface that will be used for sending and receiving the probes: If there is no interface given through the execution parameters, ZMap tries to detect the default interface by using the function `pcap_lookupdev`[4] of the Application Programming Interface (API) `Packet Capture (pcap)` (see section 3.3 Receiving Component for a more detailed description).

- Specifying the IPv4 address of the determined interface that will be placed in the probing packets as a source address: This can be done either by declaring an execution parameter manually or by using the IPv4 address detection automatism of ZMap. This automatism establishes a new IPv4/User Datagram Protocol (UDP) socket and requests its properties that were automatically attached by the Operating

[4] `http://linux.die.net/man/3/pcap_lookupdev`

System (OS). The request is made through a input/output control (ioctl)[5] system call and asks the OS for the IPv4 address of the committed socket.

- Discovering the default gateway and detecting its Media Access Control (MAC) address: The discovery of the gateway is done by sending a `GET_ROUTE` message over a Netlink socket to the Kernel. A Netlink socket is an interface between a network process and the kernel module that also supports asynchronous communication and multicasting [49]. Because of its advantages, it is intended to replace system calls like ioctl by Netlink requests in the future. The response of this request comes from the Kernel and contains the IPv4 address of the gateway. After that, a second Netlink message (`RTM_GETNEIGH`) is sent to the Kernel to request the complete ARP table of the OS. ZMap iterates through all entries of the table and compares their IPv4 addresses to the IPv4 address that was delivered before by the Kernel. The ARP table entry that matches also contains the wanted Ethernet address of the default gateway. The Ethernet address of the gateway will be used later as layer 2 destination address of the crafted probing packets.

- Building up the blacklist and parsing the destination network: The blacklist helps ZMap to exclude networks or single hosts from a scan. The structure of the blacklist is inspired by a Patricia trie. A Patricia trie is an improvement to a one-bit trie and is commonly used to build a routing table. The main idea of a Patricia trie is that a node that has only one child can be removed. This improves look up performance [50]. Every single node of the blacklist can have one of two possible statuses: `ADDR_ALLOWED` includes the IPv4 address to the scan that can be derived from the node by traversing the trie; `ADDR_DISALLOWED` excludes the derived IPv4 address from the scan. Depending on whether the user uses a whitelist to scan explicit target networks or a blacklist to exclude target networks from the Internet scan, the root node of the trie will be marked as allowed or disallowed at the initialization phase. Before a probing packet is crafted in the scanning process, ZMap looks up the calculated destination IPv4 address in the blacklist. If the node or parent-node, that includes the destination IPv4 address, contains the status `ADDR_DISALLOWED`, the calculated address is going to be discarded and the packet will not be sent.

- Finding a new primitive root of the multiplicative group: To guarantee a new permutation of the target address space for each scan, the Setup Component has to generate a new primitive root for the multiplicative group (see section 3.4 Address Generation) and has to find a random starting address [4]. The group is chosen by the number of target IP addresses, which also has to be calculated by the the Setup Component, respectively by the blacklist.

The third task is to trigger the initialization of the Sending and the Receiving Component. Furthermore the Setup Component has to start the threads of both other components.

[5] `http://man7.org/linux/man-pages/man2/ioctl.2.html`

The number of started threads depends either on the optionally given execution parameter or on the number of cores the Central Processing Unit (CPU) consists of.

3.2 Sending Component

The Sending Component can be divided into two parts; the initialization and the actual run of the scan. Among other things, the initialization function is responsible for the global initialization of the Probe Module (`global_initialize`). This means, that the Sending Component first checks if there is a function for the global initialization in the selected probe implemented. If so, it calls this function and passes the variable of the struct `state_conf`. This enables the Probe Module to access and to use the configuration parameter set of ZMap. Moreover, the initialization function has to concert the user specified bandwidth to the packet rate. This is done by taking the specific packet length of the probe into account and by calculating the sending rate, besides it is also defined by the user explicitly via an execution parameter. Next the source hardware address has to be discovered. Therefore a dummy socket is created and from it, the hardware address is extracted by using again an ioctl system call. The address is also stored as a part of the `state_conf` for later use.

The sending process itself starts by establishing a new socket for each sending thread that was calculated or specified before. The specific configuration of each socket is done in each thread separately and is twofold. First, the interface index, that is mapped to the stored interface name, and the source IPv4 address of the configured source interface have to be set. This IPv4 address actually will not be used as source address for the probing packets, but it is required for setting certain socket options. Second, the hardware address of the gateway is declared as destination address for later use as parameter of the `sendto` function [6].

Before the actual sending process is able to start, the Sending Component initializes the Probe Module; one time per sending thread (`init_perthread`). After that, the following steps are done iteratively for each packet that is sent within one sending thread:

- Adapting the timing to hit the target rate: Depending on the calculated target rate which is related to the declared bandwidth, the sending thread is sent to sleep for certain milliseconds. This delay controls the number of packets that are sent per second.
- Stopping the sending thread if one of the following checks returns `true` :
  - The Receiving Component is marked as complete.
  - The optionally specified maximum number of targets is reached.
  - The maximum runtime of the execution is reached.

[6] `http://linux.die.net/man/2/sendto`

  - The list of destination addresses, that are allocated to the thread, is completed.

- Getting a random destination IPv4 address and check if it is white- or blacklisted in the blacklist.
- Creating validation bytes by using the source and destination IPv4 addresses to identify incoming packets as correct response of a target device.
- Calling the function `make_packet` of the Probe Module to build the packet.
- Finally sending the packet.

3.3 Receiving Component

To be able to analyze incoming packets, ZMap uses an implementation of `pcap`; `pcap` is a common known API of the programming library `libpcap`[7] and enables network analyzing tools like tcpdump [8] or Wireshark[9] to capture the network traffic on a network interface. The main features of the library are [51]:

- Network traffic can be captured from several network medias like Ethernet, serial lines and virtual interfaces.
- The programming interface looks the same across every supported platform.
- The traffic can be filtered by using the provided OS Kernel module for better performance.

Depending on the used platform, the API creates a virtual device and allows the userspace to grab the captured packets from there. The disadvantage of the library is the unreliability in capturing all packets that are sent over the interface. Luce Deri [51] warns, that the precision of capturing packets differs significantly, depending on the hardware the host consists of, the used OS and the number of packets that are expected to be transferred per second over the observed interface.

Tests of ZMap showed, that ZMap is able to handle scans at gigabit line speed even while using the library libpcap. This is because in relation to the sent packets, only a small number of hosts is expected to respond. The tests also proved that, in combination with the default Linux Kernel, ZMap is able to send at 97% of the theoretical maximum speed for gigabit Ethernet. This is another argument against the use of special network drivers or an separate Kernel module [4].

Nevertheless, 10 gigabit Ethernet connections are offered more often even by cloud providers and institutes around the globe also use such a high speed connection more

[7] `http://www.tcpdump.org/manpages/pcap.3pcap.html`
[8] `http://www.tcpdump.org/manpages/tcpdump.1.html`
[9] `https://www.wireshark.org`

often. Therefore, ZMap has been improved in the last years in many respects. Regarding the Receiving Component (and the Sending Component) it supports the use of a PF_RING socket in addition to the libpcap implementation [43]. In context of this master thesis, the libpcap approach is fast and reliable enough to get results, especially as we expect only a low number of responses of for IPv6, but because of completeness the PF_RING approach is discussed in the following parts.

PF_RING is a type of socket that was developed to enable high speed packet capturing, filtering and analyzing. It is available in default Linux Kernels since version 2.6.32, that enables the use without patching the Kernel. The Kernel module can be loaded by the network analyzing tool and no specialized network device driver have to be installed [52].

In contrary to pcap, packets are not queued into Kernel network data structures. Instead incoming packets are copied directly to a circular buffer by the network adapter. Then this buffer is exported to the userspace. For new incoming packets, the Kernel moves the write pointer forward to the next address of the ring. Similar to that, the userspace moves the read pointer to the next address. Thereby, neither memory has to be allocated or freed nor packets have to be handled by upper layers; they are discarded after they were copied to the ring and the userspace can access them without the overhead of system calls. These characteristics of the PF_RING socket improves the capturing performance significantly [51].

To run ZMap in PF_RING mode, a compatible Intel 10 Gbps Ethernet NIC and Linux is required. Furthermore PF_RING has to be built and installed correctly; the Kernel module and the headers have to be installed to the correct location on the host machine [53]. The configuration of ZMap has to be done carefully: the threads have to be pinned to different physical cores and the interface has to be specified as zero-copy interface, which means that during message transmission over a network interface, there is no data copy among any memory segments [54]. In addition, David Adrian warns, that sending over 14 million packets per second - which is a lot of traffic - is not trivial and could harm network nodes. Therefore it is very important to follow scanning best practices during the research [53].

Implementation

The implementation differs slightly between the pcap and the PF_RING approach. In this master thesis, the focus is set on the pcap approach, because it is sufficient regarding the performed observations.

The Receiving Component consist mainly of one thread that is started by the Sending Component. At the beginning (recv_init()) a packet capture handle that enables ZMap capturing the network packets is obtained. The interface is set to promiscuous mode though. To reduce the number of captured packets and to save system resources, it tries to compile and apply a pcap filter [10] that is specified in the Probe Module (see section 3.5 Probe Module). To exclude duplicate responses from output economically,

[10] http://www.tcpdump.org/manpages/pcap-filter.7.html

a paged bitmap is initialized. This bitmap stores every single IPv4 address which is included in a replying packet only once.

To handle the incoming packages, the Receiving Component enters a loop until the receiving conditions, like duration of the run, number of responses, number of sent probes or a finished full scan, are fulfilled. The incoming packets are handed over to the handler that extracts its source and destination IPv4 addresses, calculates the validation checksum and forwards all obtained data to the Probe Module for further processing. The Probe Module checks if the received packet is a valid response to one of its probes and returns the result to the Receiving Component. Finally, the Receiving Component triggers an update of the Output Module by handing over the fieldset containing data that are relevant for the Output Module.

3.4 Address Generation

To ensure a scan that is done by ZMap will not overload and harm target notes, the target addresses are generated in pseudo-random order. The order can be different for every scan. An implementation a of cyclic multiplicative group ensures this pseudo-random and complete permutation. This approach comes with good performance and reduces the used memory because the address that was already scanned did not have to be stored, but ensures to traverse the complete address space. To perform a full Internet scan with 2^{32} addresses, ZMap iterates over the multiplicative group of integers modulo a prime p, where p is chosen slightly larger than 2^{32}; in case of a 2^{32} scan the next prime is $2^{32}+15$[4].

By choosing p as prime, ZMap is able to generate every possible IPv4 address except 0.0.0.0, which is specified as non-routingable IP address [55]. For each scan a new primitive root of the multiplicative group is generated and a pseudo-random starting address is chosen; this ensures a different order of target IP addresses for each scan [4].

An extension of ZMap, that has already been implemented, allows ZMap to generate the addresses in parallel to improve performance and to support sending over multiple threads simultaneously. This is realized by the development of a sharding mechanism. A shard partitions the target address space into equal amount of IPv4 addresses, that can be iterated over independently from other shards. This allows to generate the addresses within multiple threads in one ZMap process or split the execution across multiple machines in a distributed scanning mode [43].

Implementation

The cycle is generated by the iterator which is also initialized by the Sending Component. First the correct cyclic group from a given list is chosen, depending on the number of target addresses. Each group contains the prime, that is slightly larger than the number of IPs in the whitelist, a known prime root, the prime factors and the number of prime factors. A new prime root is found by generating a pseudo-random candidate, incrementing it iteratively and by checking if this candidate is coprime to the prime p

of the group -1 (p - 1). With this new prime root, the isomorphism from $(\mathbb{Z}/p\mathbb{Z})^+$ to $(\mathbb{Z}/p\mathbb{Z})^*$ is performed by using the GNU Multiple Precision Arithmetic Library (GMP) which provides arithmetical functions for large and precise numbers [11].

The shards are also initialized by the iterator. The number of shards depends on the calculated or the predefined number of threads. The initialization of each shard starts with figuring out its multiplication factor. If only one shard is declared, the multiplication factor would be the calculated prime root (generator of the multiplicative group). With more than one shard (n shards) the factor is $f = g^n$. Then the number of maximum targets per shard and the thread ID is set.

During the sending process, the Sending Component request the next element from the related shard. The shard calculates the next integer value of the multiplicative group and looks up the generated index in the blacklist. If the state is not blacklisted, the current value of the shard is set to the calculated integer value. This number is then returned to the Sending Component. The Sending Component converts this integer value to a valid IPv4 address and uses it as next target IPv4 address for scanning.

3.5 Probe Module

The integration of probes in ZMap is modular. This allows developers simple adaption of existing probes and the development of new probes for new observations and more protocols. It makes ZMap easily extensible. The required OSI layer two and OSI layer three header is provided by the ZMap core. The core also provides an empty buffer for the Probe Module. The Probe Module itself is responsible for two tasks. First, it has to fill the provided buffer with the host-specific values and second it has to validate incoming packets that where not filtered by the core. It determines if a received packet is a correct response to a probe and provides the required information to the Output Module [4].

A new probe has to provide a set of properties and callback functions that allow to connect the probe to the ZMap core. For a better overview, the names of the required callback functions can be renamed within a new Probe Module, but they have to be mapped to the generic function names. Thereby the ZMap core calls the function by its generic name, no matter which Probe Module is actually loaded.

Because this thesis focuses on scans with the Internet Control Message Protocol (ICMP), the following description deals mainly with implementation of the `ICMP_ECHO_REQUEST` Probe Module, which has been provided since the first ZMap release. This concerns especially the description of the callback functions.

A Probe Module is defined by the following properties:

- `name`: The name of the probe.

[11] `https://gmplib.org/manual/index.html#Top`

- `packet_length`: The length of a probe packet.
- `pcap_filter`: Defines the pcap filter, that allows the core to capture only packets that are relevant for the scanning process.[12]
- `pcap_snaplen`: The Maximum packet length of incoming packets that have to be captured.
- `port_args`: If set to 1, ZMap forces to specify a target port.
- `fields`: The field definitions of this Probe Module.
- `numfields`: The number of fields.

Furthermore, the following functions have to be provided and mapped.

- `thread_initialize`: Has to be provided optionally, is called at scanner initialization and forwards the struct `state_conf` to the Probe Module. This allows the Probe Module to access the global configuration of the scanner.
- `thread_initialize`: Is called by each sending thread and provides the declared buffer. It divides the buffer into the different layer sections and fills in the layer 2 information, which is known at this moment: the source hardware address and the gateway hardware address.
- `make_packet`: Is called once per target address and updates the crafted packet by the missing values: Source IPv4 address, destination IPv4 address and validation bytes. In case of the Probe Module `ICMP_ECHO_REQUEST`, the function crafts the ICMP header and and fills in the required ICMP header values. After that, it calculates the IPv4 checksum by calling a function which is provided by the scanner core. Then the checksum is added to the IPv4 header.
- `print_packet`: The function is called if ZMap is executed in dry-run mode. Instead actually sending the crafted packet, this function prints the content of the packet and the headers in a human readable format to the default output.
- `validate_packet`: Is called by the Receiving Component for each captured packet. This function calculates the validation bytes and compares them to the validation bytes that are sent in the payload of the captured packet. If they match, the core continues with the processing of the packet.
- `process_packet`: Is called by the Receiving Component for each packet which is captured and identified as correct response to a probe. The function extracts information, that classifies the packet. In case of a reply of an `ICMP_ECHO_REQUEST` probe, it adds the ICMP header values to the fieldset and determines what type of reply was returned (e.g. echo reply, unreachable, time exceeded).

[12] `http://www.tcpdump.org/manpages/pcap-filter.7.html`

- `close`: Is used to cleanup the probing data and is called when the scanner terminates.

3.6 Output Module

The Output Module also provides a modular structure. The basic built-in output module includes simple text output, that contains a list of IPv4 addresses of hosts that response to probes. But to extend the output and post processing, new output modules can be developed and registered. ZMap already provides such extended modules. They bring the output to different formats like Comma-separated values (CSV) or JavaScript Object Notation (JSON). The callbacks of an output module are triggered by several events at different places at the scanner core. This allows ZMap to print more than just the result of the scan.

3.7 Summary and Conclusion

Static code analysis in combination with dynamic code analysis enabled us to understand how ZMap works. The result of the analysis is a new model, that describes ZMap in six components. This model is used later to identify the parts that have to be extended to make ZMap support IPv6 and is assisting during the coding process. The tasks of each component are described in detail and their main functions are highlighted. The six identified components are called: *Setup Component*, *Sending Component*, *Receiving Component*, *Address Generation*, *Probe Module* and *Output Module*.

CHAPTER 4

Adaptions of ZMap

The technical aim of this thesis is making ZMap IPv6 capable. This requires adaptions of ZMap in almost every module that is described in the chapter before (3. Static and Dynamic Code Analysis of ZMap). The new implementations should extend ZMap, which means that scans of IPv4 addresses must be also possible. This is accomplished by checking whether ZMap is executed in IPv6 mode. If the IPv6 mode is enabled, it has to call especially implemented, IPv6 focused functions that exist in parallel to the IPv4 related code. With regard to IPv6 the focus is set on the Internet Control Message Protocol version 6 (ICMPv6) support, because it takes an important role in the concept of IPv6. It allows to discover networks and collect information about connected nodes. This chapter describes the most important and considerable adaptions, which can be also relevant for IPv6 support for other programs written in C. Figure 4.1 provides an overview about the parts that had to be adapted (yellow bordered) and parts that had to be developed completely new (red bordered).

4.1 Setup Component

The following adaptions were made in the Setup Component.

4.1.1 Adding Parameters

To add a new parameter to ZMap, the file `zopt.goo` had to be edited by adding a new name and and the desired parameter description. The name of the option declares the execution parameter. This parameter can be shortened optionally and is written next to the name. After that follows the description of the parameter. This description is displayed if ZMap is executed with the parameter `--help`. The next line shows what kind of datatype the parameter expects and if the parameter is optional or mandatory for execution.

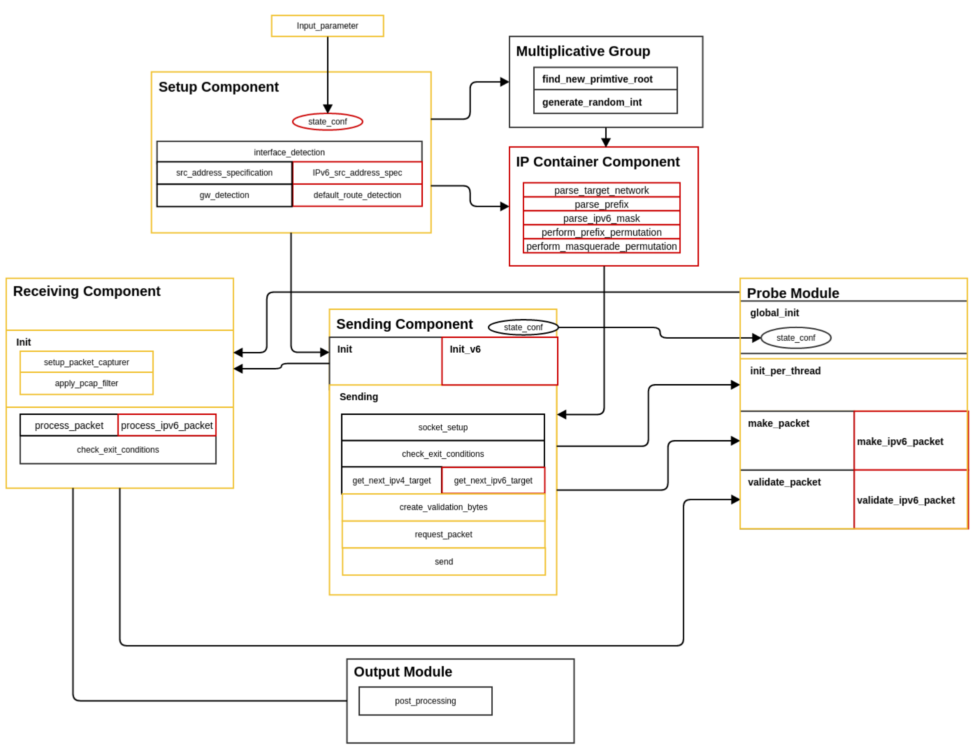


Figure 4.1: ZMap adaptions in technical point of view

To support IPv6, several new execution parameters had to be added and a new section was inserted to declare the IPv6 options. Following lines were added to the file `zopt.goo`:

```
section "Ipv6 options"
option "ipv6"                    6 "Run ZMap in IPv6 mode"
    optional
option "addr32"                  - "Use IPv6 implementation"
    optional int
option "ipv6mask"                - "Mask for IPv6 permutation"
    typestr="mask"
    optional string
```

Description of the new parameter in more detail:

`option "ipv6"`: This parameter tells ZMap if it should run in IPv6 mode. It is the main switch that enables branching the IPv6 code and functions out of the default IPv4 implementation.

`option "addr32"`: This parameter is optional and assists the first of the two implemented scanning approaches which are described in the upcoming sections. The expected value lies between 0 and 3, which represents one of the four 32 bit segments of an IPv6 address.

`option "mask"`: This parameter is also optional and triggers the IPv6 scanning approach that is implemented as second option. The expected parameter is a string that represents a mask in IPv6 address hexadecimal notation [?].

To make the options accessible for ZMap at execution time, they have to be generated by GNU gengetopt (3.1 Setup Component). The Setup Component stores them in the struct `state_conf`. The values can be requested by accessing the struct member that is named like the related execution parameter.

Command
The execution parameters are generated after executing the following command: `sudo gengetopt -C --no-help --no-version --unamed-opts=SUBNETS -i <path to zopt.goo> -F <path to the destination>`

Table 4.1: Generate configuration file for new execution parameters

4.1.2 Determining the Global Unicast Address of the hardware interface

The first time a code branch is required for a special IPv6 implementation is the determination of the Global Unicast Address of the specified hardware interface. This address will be used as layer three source address of a probe. This is done by creating an if-statement in front of the default function
(`int get_iface_ip(char *iface, struct in_addr *ip)`), which is responsible for determining the IPv4 source address. The statement checks if ZMap runs in IPv6 mode; if yes, it calls a new implemented function
`int get_iface_ip6(char *iface, struct in6_addr *ip)` instead of the IPv4 counterpart function. Within the IPv4 `get_iface_ip` function, an ioctl system call is executed to get the IPv4 address (3.1 Setup Component). Because ioctl system calls will be replaced in near future [49], there is no related system call available to request the corresponding IPv6 address. Instead the Kernel function `getifaddr`[1] has to be used. This function returns a linked list of structures that contain information about the local network interface [56]. This list has to be iterated over all entries and the checks described in Table 4.2 have to be performed.

[1] http://man7.org/linux/man-pages/man3/getifaddrs.3.html

	For each entry in this list, the following verification has to be done:
1	Check if the expected entry contains data in the structure member of `ifa_addr`.
2	Compare the name of the network interface of the entry to the defined source network interface.
3	Check if the address of the entry is part of the address family `AF_INET6`, which means it is an IPv6 address.

Table 4.2: Verification steps for all entries of the list returned by the function getiffaddr.

The result of the checks is an IPv6 address in the form of a structure called `in6_addr`, which is defined in the Kernel header file <netinet/in.h> [57]. But there can be more than one IPv6 address assigned to a network interface. There is a list of different types of IPv6 addresses and all of them have different responsibilities (see section 2.1 Internet Protocol). For scanning an address space in the IPv6 Global Area Network (GAN), only the Global Unicast Address is relevant. The Kernel provides some macros to check what type a given IPv6 address belongs to (see Table 4.3). Unfortunately there is no macro that checks if the given IPv6 address is a Global Unicast Address. Instead it is mandatory to check all other IPv6 address types. Only if all these checks are negative, it can be assumed that the examined address is a Global Unicast Address.

	The following macros are provided by the Kernel, to determine the type of an IPv6 address:
1	`IN6_IS_ADDR_LINKLOCAL`: returns true if the address is a Link Local IPv6 address
2	`IN6_IS_ADDR_SITELOCAL`: returns true if the address is a site local IPv6 address, which is deprecated according to RFC 3879 [58]
3	`IN6_IS_ADDR_V4MAPPED`: returns true if the address is an IPv4-mapped IPv6 address
4	`IN6_IS_ADDR_V4COMPAT`: returns true if the address is an IPv4-compatible IPv6 address
5	`IN6_IS_ADDR_LOOPBACK` returns true if the address is an IPv6 Loopback address
6	`IN6_IS_ADDR_UNSPECIFIED` returns true if the address is an Unspecified address

Table 4.3: Macros for determining the type of IPv6 address.

4.1.3 Detecting the Gateway

For IPv6 scanning the gateway detection of the IPv4 implementation could be used, if the gateway for IPv6 connections is the same gateway as for IPv4 connections. Because only the hardware address of the gateway is required to craft the probes (layer 2 target address), it makes no difference if the IPv4 approach is used to determine the hardware address or if a new implemented function for an IPv6 connection is called. A special implementation for IPv6 gateway detection is therefore only necessary, if there are two different gateways or if there is no IPv4 connection available. In near future it can be expected that IPv4 cannot be used anymore, therefore a new approach for IPv6 gateway detection was implemented in ZMap.

When ZMap is executed in IPv6 mode, the new function `int get_default_gw6(struct in6_addr *gw, char *iface)` is called to determine the IPv6 address of the default gateway. Like in the solution of the IPv4 gateway detection, also a Netlink socket (see section (3.1 Setup Component) is established, but in contrary to the IPv4 solution the new function sends a `RTM_GETROUTE` request to the Kernel. The Kernel replies with its routing table. This routing table has to be scanned to detect the default route of the IPv6 connection. This is done by filtering first if the entry is part of the address family `AF_INET6` and second if the type of the entry is the type `RTA_GATEWAY`. The remaining entry of the table contains the IPv6 address of the gateway for the IPv6 connection.

This IPv6 address is used to resolve the required hardware address of the gateway. The new function `int get_hw_addr6(struct in6_addr *gw_ip, char *iface, unsigned char *hw_mac)` is called. This time the IPv6 implementation is nearly similar to the IPv4 solution. Again, a netlink socket is established and a `RTM_GETNEIG` request is crafted and sent to the Kernel. The crafted requests include network addresses of the family type `AF_INET6.` The response of the request contains the table with the hardware addresses and the network addresses of the network neighbors. Each entry of this table has to be compared to the IPv6 gateway address which has determined before. The matching entry contains the required hardware address of the gateway.

4.2 IPContainer Component

For the IPv6 extension of ZMap the blacklist is excluded if ZMap runs in IPv6 mode. The reason is, that in relation to the large number of IPv6 addresses and prefixes, a blacklist would not be efficient enough to exclude single IPv6 addresses from a scan. Because the blacklist builds the structure of the target addresses for a scan, and is therefore deeply rooted in the scanner core, the exclusion of the blacklist is combined with the address generation, one of the most significant distinctions to the IPv4 implementation.

At this point, a new component which is exclusively available in the IPv6 mode had to be developed. The component is called IPContainer and holds all Information about target addresses. The new component is the counterpart, that corresponds to the blacklist. It is

responsible for storing and interpreting the parsed input parameter which represents the target IPv6 network and for making the given address permutable. It is also responsible for the decision of which one of the two available scanning approaches, depending on given execution parameter, is executed. Instead of calling the concerning function of the blacklist to get a new target address, the IP Container delivers the next address.

4.2.1 Parsing the destination network, prefix and mask

For parsing the input parameter and converting it to a target IPv6 network, two different functions are available. If next to the network address the execution parameter `ipv6mask` is not set, the function
`static int ipcontainer_init_from_String(char *ip6)` is called. This function expects an IPv6 network in CIDR notation as string. The address and the prefix are cut and stored separately. Then the component tries to cast the IPv6 address string into the structure `in6_addr` and checks if the prefix is a number between 32 and 64. Only if both are valid inputs, the program continues. On the other site, if the parameter `ipv6mask` is set, the function
`static int ipcontainer_init_from_Mask(char *ip6, char *v6mask)` is called. this function ignores an optionally added prefix information and tries to cast the IPv6 address and the IPv6 mask into a structure `in6_addr`. If both casts were successful, the structures are stored and the program continues.

4.2.2 Cycle

Because a random address generation is also required for an IPv6 scan, ZMap uses the multiplicative group of the scanner core (see section 3.4 Address Generation) in IPv6 mode to generate integers values randomly. The original implementation supports number generation up to 2^{32} values, which is equal to the number of possible IPv4 addresses. This allows ZMap to swap the complete content of an IPv4 address space within a single run. The concept of IPv6 provides 2^{128} unique network addresses. It would take too much time to scan and permute all possible IPv6 addresses. Therefore both implemented scanning modes of the IPv6 extension, namely Prefix Permutation and Masquerade Permutation, support also address generation only up to 2^{32} unique addresses. Because of this decision, the default implementation of the random value generation can be used. But as described in section 3.4 Address Generation, the behavior of a multiplicative group excludes zero values from value generation. This is relevant for an IPv4 run, because the address 0.0.0.0 is not a valid target address. In the extension of IPv6 this is a problem, because in both scanning approaches, a part in the defined IPv6 address persists while the other part permutes. This means that ZMap has to calculate zero values for the permuting part to cover all addresses that have to be scanned. To solve this problem, the value which is generated by the `cycle` is decremented by 1. The performed tests proved, that this measure enables ZMap to generate all required IPv6 addresses and it still uses all benefits of the multiplicative group.

4.2.3 Prefix Permutation

The intention of the first scanning approach was to scan prefixes of potentially deployed IPv6 networks. For each network, which is defined by a prefix of the length of 64 bits, one address is generated and a crafted packet is sent to the target host within this network. The idea is that depending on the response, ZMap receives (or does not receive), statements can be made about the global routing structure. This is realized by specifying the length `n` of the second 32 bits of a prefix. The result, which has a maximum length of 32 bits and is called the `subnet ID`, will be permuted and one target address will be generated for each subnetwork (Table 4.4).

128-bit Ipv6 Address		
32 + n bits	32 - n bits	64 bits
global routing prefix (fixed)	subnet ID (permuted)	interface ID (fixed)

Table 4.4: IPv6 Address Structure with Variable Prefix Length

To perform a scan, ZMap has to start in IPv6 mode (input parameter `-6`). ZMap then expects the IPv6 address and the prefix in CIDR notation. The prefix has to be defined between the values 32 and 64 to be able to calculate the correct permutation range. ZMap takes the smallest possible subnetwork with a prefix length of 64 bits and subtracts the specified prefix length. This defines the variable part of the network address. E.g.: If an address is given with an 32-bit prefix, the maximum number of addresses will be generated, namely $2^{(64\text{-}32)} = 2^{32}$addresses (Table 4.5). The number of target addresses is stored separately in a variable of the IP Container and can be read by any other component. Among others, the number is required to initialize the cycle to define the largest integer value that will be calculated.

Input: 2a02:8388:8fa2:214c:c85e:8513:bdb4:79ef/32							
32 bits		32 bits		32 bits		32 bits	
16 bits	16 bits	16 bits	16 bits	16 bits	16 bits	16 bits	16 bits
2a02	8388	8fa2	214c	c85e	8513	bdb4	79ef
Fixed		Permuted from 0000:0000 to FFFF:FFFF		Fixed			

Table 4.5: IPv6 Address Permutation with 32-bit Prefix

The larger the prefix is defined, the smaller is the address space that will be scanned. E.g.: If the input address is specified with an 48-bit prefix, the first 48 bits of the address are fixed and only the following 16 (64-48) bits will be permuted (Table 4.6). From this follows, that an input address that is specified with an 64-bit prefix can be seen as host address, which means only one target address will be generated.

Input:2a02:8388:8fa2:214c:c85e:8513:bdb4:79ef/48							
32 bits		32 bits		32 bits		32 bits	
16 bits	16 bits	16 bits	16 bits	16 bits	16 bits	16 bits	16 bits
2a02	8388	8fa2	214c	c85e	8513	bdb4	79ef
Fixed			Permuted from 0000 to FFFF	Fixed			

Table 4.6: IPv6 Address Permutation with 48-bit Prefix

To enable the IPContainer Component to generate the new target addresses with low utilization of processing power, the address crafting is inspired by the concept of IPv4 subnetmasking. Bitwise operations in C are fast and save resources [59], therefore they are used to mask the given IPv6 address as first step and merge them with the integer value that is calculated by the multiplicative group of the scanner core as second step.

Depending on the requirements, C provides several members of the structure `in6_addr`. This members deliver the IPv6 address in form of arrays of different size and datatypes:

- `uint8_t __u6_addr8[16]`
- `uint16_t __u6_addr16[8]`
- `uint32_t __u6_addr32[4]`

This allows the program to split the address easily and focus on the required parts. At the first implementation of the described scanning approach, only the second 32 bits of the IPv6 address have to be manipulated. Therefore only the second array entry (`[1]`) of the member `__u6_addr32[4]` is used. Depending on the specified prefix length, a part of this 32 bits contains information that is not required anymore, because it has to be substituted by the integer value that is generated by the multiplicative group. To erase this part, a bit mask is created. This mask starts with as much `1` values as bits stay fixed in the 32-bit array entry. After the fixed part, the mask is filled up with `0` values. The AND-Operator is than used to operate on the second array entry and the crafted mask. This leaves the required part of the 32 bits untouched and replaces the not required part with `0` (Table: 4.7).

To get the the complete crafted and random target address, the integer value, that is calculated by the multiplicative group, has to be merged with the masked IPv6 address. This is done by using the XOR-Operation (Exlusive-OR) that ensures again that the fixed part of the second 32 bits stays unmodified (Table: 4.11). The output is the IPv6 address within the specified address space, that can be used as target network address.

The first test scans showed, that more flexibility in defining the target address space is required. Therefore, the first scanning approach was extended by a new execution

<table>
<tr><td colspan="32">Input: 2a02:8388:8fa2:214c:c85e:8513:bdb4:79ef/48</td></tr>
<tr><td colspan="32">32 bits</td></tr>
<tr><td colspan="16">16 bits</td><td colspan="16">16 bits</td></tr>
<tr><td colspan="16">8fa2</td><td colspan="16">214c</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td colspan="32">MASK GENERATION</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td colspan="32">AND Operation</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td colspan="16">8fa2</td><td colspan="16">0000</td></tr>
<tr><td colspan="32">Output: 2a02:8388:8fa2:0000:c85e:8513:bdb4:79ef/48</td></tr>
</table>

Table 4.7: Mask 16 Bits of the Second 32 Bits with Prefixmask

<table>
<tr><td colspan="32">Input: Output from Masking: 2a02:8388:8fa2:0000:c85e:8513:bdb4:79ef/48</td></tr>
<tr><td colspan="32">32 bits</td></tr>
<tr><td colspan="16">16 bits</td><td colspan="16">16 bits</td></tr>
<tr><td colspan="16">8fa2</td><td colspan="16">0000</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td colspan="32">RANDOM INTEGER GENERATION (e.g. $41266_{(10)}$)</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td colspan="32">XOR Operation</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td colspan="16">8fa2</td><td colspan="16">a132</td></tr>
<tr><td colspan="32">Output: 2a02:8388:8fa2:a132:c85e:8513:bdb4:79ef</td></tr>
</table>

Table 4.8: Merge Random Value and Masked IPv6 Address

parameter: `addr32`. The parameter expects an integer value between `0` and `3`. It represents the index of the array that splits the structure `in6_addr` into four 32-bit parts. The parameter allows to define which one of the array entries should ZMap use for masking and permutation. The parameter is optional; if it is not given, the variable behind is set to `1` as default and the behavior of the address generation remains as described in this section. The disadvantage of this extension is, that with regards to content changing, the index of the array makes the specification of the prefix obviously unpinning and the flexibility does not increase significantly.

4.2.4 Masquerade Permutation

Furthermore tests (see chapter 6 Results) showed, that even the extended version of the first scanning approach (4.2.3 Prefix Permutation) does not provide the flexibility that is required to gain significant results. To equip ZMap with a more adjustable way to define the target address space, a new scanning approach was developed. The idea is

to define an IPv6 mask that is also inspired by a subnetmask of an IPv4 network. For the setup definition of a scan, the mask is extended to a length of 128 bits. The value 1 in this masks marks a bit in the target address space as fixed; a 0 value marks the corresponding bit in the target address space as replaceable.

Therefore a new input parameter `ipv6mask` is provided. It expects a string in the structure of an IPv6 address in human readable notation (see section 2.1.2 IPv6). ZMap tries to cast this string into a `in6_addr` structure. On failure, the program stops. The first task is to calculate the number of required target addresses to initialize the cyclic. Compared to the first scanning approach, there is no address prefix to calculate this number. Instead the number of 0 values in the mask has to be counted. The number of target addresses is then 2^n, where `n` stands for the number of 0 values in the mask. The mask is divided into the four already described members `__u6_addr32[4]` of the structure `in6_addr`. The bits are counted for each of this members separately. For optimization purposes, the IPContainer Component stores the index of the member which contains 0 values for later use. The bitcount function itself uses the idea of the item number 169 of the HAKMEM, which is also known as AI Memo 239 of the Massachusetts Institute of Technology. Written in the year 1972, it contains a number of useful algorithms [2]. Item number 169 contains ten instructions of the machine language PDP-10, that counts the ones in a word with length up to 62 bits [60].

The approach allows to count the bits with constant time and constant memory by using the concept of parallel counting [61]. The internet provides several implementations of the HAKMEM counting in different programming languages [3]. Because the HAKMEM counting counts only the 1 in a word, the result has to be subtracted from 32 for each 32-bit structure member to get the required number of zeroes.

```
static int calc_number_masking_Bits(uint32_t u){
    unsigned int uCount;
    uCount = u - ((u >> 1) & 033333333333) - ((u >> 2) & 011111111111);
    return (32-(((uCount + (uCount >> 3)) & 030707070707) % 63));
}
```

Code Snippet 4.1: HAKMEM Bitcounting [61]

The next task is to mask the given IPv6 address. This time, the mask does not have to be generated. It can be used in the form as it has been declared at the related input parameter. As described in table 4.9, the AND-Operation is used to merge each 32-bit part of the specified IPv6 address with the corresponding 32-bit part of the IPv6 mask.

[2] https://dspace.mit.edu/bitstream/handle/1721.1/6086/AIM-239.pdf?sequence=2

[3] http://blog.hupili.net/p--20130328-bit-counting/

The result is a new IPv6 address, where all bits of the old IPv6 address, that are not masked by the IPv6 mask, are set to 0. This new address is stored in the IPContainer Component and is used as basis to generate the complete set of target IPv6 addresses.

Input: IPv6 Address: 2a02:8388:8fa2:214c:c85e:8513:bdb4:79ef Input: IPv6 Mask: ffff:ffff:ff00:f00f:ffff:0fff:ffff:ff00							
32 bits		32 bits		32 bits		32 bits	
16 bits	16 bits	16 bits	16 bits	16 bits	16 bits	16 bits	16 bits
IPv6 Address							
2a02	8388	8fa2	214c	c85e	8513	bdb4	79ef
IPv6 Mask							
ffff	ffff	ff00	f00f	ffff	0fff	ffff	ff00
AND-OPERATION							
2a02	8388	8f00	200c	c85e	0513	bdb4	7900
Output: Masked IPv6 Address: 2a02:8388:8f00:200c:c85e:513:bdb4:7900							

Table 4.9: IPv6 Address Masking with IPv6 Mask

The next part follows after a new integer value has been calculated by the multiplicative group. This integer value has to be spread over the base IPv6 address. The single bits of the integer have to be inserted at the places which where set to 0 during the masking step before (Table 4.10).

Input: Masked IPv6 Address: 2a02:8388:8f00:200c:c85e:513:bdb4:7900 Input: Generated Integer: $(626E843)_{(16)}$							
32 bits		32 bits		32 bits		32 bits	
16 bits	16 bits	16 bits	16 bits	16 bits	16 bits	16 bits	16 bits
Masked IPv6 Address							
2a02	8388	8f00	200c	c85e	0513	bdb4	7900
Integer Value							
$(6\ 2\ 6\ E\ 8\ 4\ 3)_{(16)}$							
GENERATE new IPv6 Target Address							
2a02	8388	8f62	26Ec	c85e	8513	bdb4	7943
Output: Target IPv6 Address: 2a02:8388:8f62:26ec:c85e:8513:bdb4:7943							

Table 4.10: Inserting the Random Generated Integer Value

To carry this out, a new algorithm that crafts a new IPv6 mask, called inserting mask, was developed (algorithm 4.1). To increase the performance, this algorithm uses the indices that were stored before in the IPContainer and inspects only the 32-bit parts of the origin mask that contains 0 values. If a 32-bit segment of the origin mask contains only 1 values, all bits of the corresponding segment of the inserting mask are set to 0 in one step. The remaining segments are crafted bit by bit, by comparing the bits of the origin mask and the bits of the random generated integer value. In the end the inserting

mask contains the bit information of the random generated integer value, spread over 128 bits and inserted at at the required positions of the corresponding 32-bit segments.

Algorithm 4.1: Inserting Mask Creation

```
   Input: OM[4][32] = 32-bit segments of the origin mask , R[32]=Random Generated
          32-bit Integer value
   Output: IM[4][32] = 32-bit segments of the inserting mask
 1 IM[1-4][1-32] ← 0;
 2 R_index ← 1
 3 for Segment_index ← 1 to 4 do
 4 |  IM_index ← 1
 5 |  for Bit_index ← 1 to 32 do
 6 |  |  if OM[Segment_index][Bit_index] = 1 then
 7 |  |  |  IM_index ← IM_index + 1;
 8 |  |  |  break for;
 9 |  |  end
10 |  |  if R[R_index] = 1 then
11 |  |  |  IM[Segment_index][IM_index] ← 1;
12 |  |  |  break for;
13 |  |  end
14 |  |  IM_index ← IM_index + 1;
15 |  |  R_index ← R_index + 1;
16 |  end
17 end
18 return IM;
```

To get a new target address, the last step is to merge all 32-bit segments of the masked IPv6 address with the corresponding 32-bit segments of the inserting mask. Table 4.11 shows a more detailed view of this operation based on the merge of the second 32-bit segments.

The creation of the inserting mask and the ensuing merge of the masked IPv6 address and inserting mask has to be done for each output of the multiplicative group to get new target addresses. The advantage of this approach is the highest form of flexibility. The option to mask single bits everywhere in the specified target address space enables to scan complex patterns with no restriction. For performance purposes, only one limitation that checks that a maximum of 32 bits are selected for permutation is implemented.

4.3 Sending Component

The Sending Component is extended by an exclusive function for the initialization in IPv6 mode. This function triggers the initialization of the iterator that in turn initializes the IPContainer instead of the blacklist. Except some IPv6 validations of the IPv6 source

<table>
<tr><td colspan="32">Input: Masked IPv6 Address: 2a02:8388:8f00:200c:c85e:513:bdb4:7900
Input: Inserting Mask: 0000:0000:0062:06E0:0000:8000:0000:0043</td></tr>
<tr><td colspan="32">32 bits</td></tr>
<tr><td colspan="16">16 bits</td><td colspan="16">16 bits</td></tr>
<tr><td colspan="16">8f00</td><td colspan="16">200c</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td colspan="16">0062</td><td colspan="16">06e0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td colspan="32">XOR Operation</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td colspan="16">8f62</td><td colspan="16">26ec</td></tr>
<tr><td colspan="32">Output: Target IPv6 Address: 2a02:8388:8f62:26ec:c85e:8513:bdb4:7943</td></tr>
</table>

Table 4.11: Detailed View: Merge Inserting Mask and Masked IPv6 Address

address, the remaining parts stay the same as the related function of the IPv4 mode implementation (see 3.2 Sending Component).

The function `send_run` which is responsible for the sending process (also introduced in section 3.2 Sending Component), is extended by two conditional branches. The first branch allows the Sending Component to skip the look-up in the blacklist and to obtain the first generated integer value directly from the multiplicative group. The second branch is located at the end of the component. Within this branch, the following tasks are performed:

- Casting the IPv6 source address from the string stored in the `state_conf` to a valid structure `in6_addr`.
- Requesting the IPv6 destination address from the IPContainer Component.
- Generating the validation bytes by selecting a 32-bit segment of the destination address.
- Calling the function `make_ip6_packet` of the probe to craft the final packet.
- Sending the packet.

4.4 Receiving Component

The Receiving Component is extended by one new function called `void handle_ip6_packet`. Equivalent to its IPv4 counterpart, the function processes incoming packets. It extracts the Ethernet header and the IPv6 header. Next it calculates the expected validation bytes, by taking the IPv6 source address into account. The gathered information is forwarded to the probe module to validate the packet. If the probe module does

not declare the packet as valid, the Receiving Component discards it. Otherwise the component declares a new fieldset for the Output Module and stores the IPv6 header information in it. For classification, the packet is again forwarded to the Probe Module without expecting a response. Finally, the function performs clean up statements and triggers the update of the Output Module.

4.5 Probe Module

As described, ZMap integrates probes modular and allows to extend the functionality by developing new probes easily. To extend ZMap by IPv6 scans, the header files of the Probe Module have to be extended by several IPv6 functions. The general packet definition is extended by the function `void make_eth_header_ip6` to generate the Ethernet header and by the function `void make_ip6_header` to create the IPv6 header. In contrast to the IPv4 solution, the Ethernet header declares IPv6 as protocol of the next layer. The function for crafting the IPv6 header declares and maps the required protocol fields (see section 2.1.2 IPv6, Figure 2.2). Furthermore the packet definition contains a function that converts the content of the IPv6 header to a human readable string format. This is required to set the specified fieldsets for the Output Module.

The definition of a probe is also extended by two new functions.

- `int (*probe_make _ip6_packet_cb)`: This function is responsible for collection all information together and crafting the complete packet out of it. In contrast to the IPv4 counterpart it requires `in6_addr` structures as function parameter.
- `int (*probe_validate_ip6_packet_cb)`: This function is called by the Receiving Component. Its task is to calculate the validation bytes by using the packet specific information (i.e. payload or packetID etc.. The output is compared to the validation bytes that were calculated by the Receiving Component. Only if both byte sequences are equal, the received packet counts as correct response.

A new IPv6 probe has to implement these two new functions. For usability purposes, both IPv4 counterparts should be also implemented and should stop the program if one of them is called. This can happen if the IPv6 probe is chosen for an IPv4 scan.

ICMPv6 Probe

This master thesis focuses on scans with ICMPv6 probes. Therefore a new probe module `module_icmp6_echo` was developed. If this probe module is chosen for a run, ZMap crafts ICMPv6 requests from type `ECHO_REQUEST` (type number 128). The expected response is an ICMPv6 response from type `ECHO_RESPONSE` (type number 129). In the module all functions that are defined in the module header file are implemented. Therefore the module is able to craft ICMPv6 packets that contain all required headers: Ethernet header, IPv6 header and ICMPv6 header. The source hardware and network address and the destination hardware and network address are forwarded by the Sending

Component. The module sets the validation bytes that were also calculated by the Sending Component, into the ID field of the ICMPv6 header.

Compared to IPv4, the IPv6 header does not support an integrity check, instead the checksum field of the ICMPv6 header is used to detect data corruption in the ICMPv6 message and parts of the IPv6 header. This checksum calculation is implemented in the global packet definition and is called by the probe module. It is calculated as described in section 2.2.2 ICMPv6, by building a "pseudo-header" of some IPv6 header fields first and after that, by creating the 16-bit ones' complement of the ones' complement sum of the entire ICMPv6 message, starting with the ICMPv6 message type field [9].

The validation of a response packet is done by extracting the ID field from the packet and by comparing it to the validation bytes that where calculated by the Receiving Component. At the moment, the module validates and classifies only `ECHO_RESPONSE` packets. Hypothetically it is also possible to validate other responses to an `ECHO_REQUEST`, like `No route to destination` or `Address unreachable`. This makes it necessary to look in to the payload of the response, because ICMPv6 stores the original request packet in there. For validation, the payload has to be extracted and the validation bytes have to be calculated by the IPv6 header data of the packet in the payload.

4.6 Summary and Conclusion

The components that were identified in Chapter 3 Static and Dynamic Code Analysis of ZMap, have been adapted to make ZMap IPv6 capable. To still support IPv4, new execution parameters were added. They allow to check whether ZMap should execute the new code parts, which are only relevant for IPv6 scanning; compared to the IPv4 version, the blacklist will not be used in the IPv6 mode, because it is not pertinent for this approach. Rather a new component, the IPContainer, was developed. It is responsible for storing information and for performing calculations regarding IPv6 target addresses. ZMap is now able to scan IPv6 networks in two different ways. The prefix permutation is specialized for the analysis of network prefixes. The setup of prefix permutation is easy to configure but not that flexible. The masquerade permutation is the second scanning approach and allows to scan complex network address patterns. Furthermore a new probe has been developed, that allows to scan networks by taking ICMPv6 Echo Request messages into account.

CHAPTER 5

Setup for Observations

A correct and complete setup for an observation is most important to obtain meaningful statements of scanned networks. The accuracy of the results correlates with the quality of the analysis of target networks without actually scanning them. This chapter discusses the steps that are recommended to do before a scan is performed.

5.1 Finding Target Networks

To be able to perform scans, the target networks have to be identified and specified. There are several ways to find networks. One way is to search the Internet (Forums, Blogs etc.) for hints of IPv6 addresses in use. This way is laborious but it helps to find e.g. used DNS server or web server IPv6 addresses. It is especially important for the first testing purposes to find host addresses that are replying to scanning requests constantly. If one host address is known, it is easy to find the the related network. Each regional Internet Registry provides an online database. It contains information about the IP allocation they are responsible for. By using the service `whois`[1] the databases can be queried.

To get information about a host address in Europe, the database of RIPE NCC has to be consulted. The host address can be entered into the database query field of the homepage [2]. The result is a set of detailed information about the IP address range. A practical example: The IPv6 address of the DNS server "res1.a1.net" of the Austrian ISP A1 Telekom Austria AG can be found on several homepages that are exploring the distribution of the new Internet Protocol [3]. To get the complete address range, the known IPv6 address is inserted in the "whois" field of the RIPE NCC database. The result is a text block containing the information of Table 5.1.

[1] `https://apps.db.ripe.net/search/query.html`
[2] `https://www.ripe.net/`
[3] `http://www.allesedv.at/IPv6/host/res1.a1.net`

inet6num:	2001:850::/30
netname:	AT-TELEKOM-20020725
country:	AT
org:	ORG-TAA1-RIPE
admin-c:	TAV6-RIPE
tech-c:	TAV6-RIPE
status:	ALLOCATED-BY-RIR
notify:	ip.netdesign@a1telekom.at
mnt-by:	RIPE-NCC-HM-MNT
mnt-by:	AS8447-MNT
mnt-lower:	AS8447-MNT
mnt-lower:	AS1901-MNT
mnt-routes:	AS8447-MNT
mnt-routes:	AS1901-MNT
created:	2012-10-25T09:31:30Z
last-modified:	2016-05-19T05:49:02Z
source:	RIPE

Table 5.1: RIPE NCC "Whois" Result of A1 DNS Server

For this master thesis, the first three entries are relevant. `inet6num:` shows the complete IPv6 network which is allocated to A1. `netname:` is the name of the IP range. According to the RIPE Database documentation, it is recommended that the same `netname` is used for any set of assignment ranges that are used for a common purpose, such as a customer or a service [62]. `country`: This attribute stands for a country using the ISO 3166-2 letter country codes. According to the description in the documentation of this attribute, it can represent the location of the head office of a multi-national company as well as the location of the server center as well as the home of the end user. Due to missing specification, the location classification of an IP address range by means of this attribute is not reliable [62]. But for this thesis, such classification are necessary because it is focused solely on Austrian IPv6 ranges. Fortunately, the database contains more attributes that are not shown by default in the output of "Whois". They are described later on.

The result of this first analysis is a network with a 30-bit prefix that belongs to the mentioned Austrian ISP. Because this network is too large for a complete scan, it has to be analyzed in further detail. The network can be divided into four smaller 32-bit sub-networks (2001:850::/32 - 2001:853::/32). With the help of the IPv6 address of the discovered DNS server (2001:850:3010:101:101::101), its related 32-bit sub-network can be identified as 2001:850::/32. By defining an address or prefix pattern (see section 5.2 ReferencesRecognizingPatterns), the segment is ready for scanning. At least one response must be found; namely the already known DNS server; but hopefully more than this one.

The second and more comfortable way to collect IPv6 address ranges is to query the RIPE database directly. There are several filter parameters available that can be looked up in the RIPE Whois Database Query Reference Manual[4]. To discover Austrian IPv6 addresses, the database first of all has to be filtered for the country code "AT". Unfortunately, the attribute `country` is a non queryable attribute. The service "whois" denies the filter `-i country AT"` with the message "ERROR:105: attribute is not searchable. "country" is not an inverse searchable attribute." On the other side, if the query string contains the known name of the organization A1 Telekom Austria AG, the result shows detailed information about the organization, but not the required IPv6 addresses and netnames.

To get the required information anyway, RIPE NCC offers their complete database for download. The RIPE File Transfer Protocol (FTP) server [5] provides a lot of different files which contain the complete database or only parts of it. To get the IPv6 address space that is registered at RIPE NCC, only the file `ripe.db.inet6num.gz` [6] is relevant. The compressed file contains the database as a large text file. The file is too large for reading in a casual text editor, but by using the Unix command `more`, the file can be read line by line (Table 5.2).

Command
`more ripe.db.inet6num`

Table 5.2: Read content of ripe.db.inet6num

The content of the file represents blocks that group the data by the attribute `inet6num`. One block represents an IPv6 range and lists all its attributes (Table 5.3).

The file consist of 9119738 lines, therefore reading the file line by line manually takes time and is inconvenient. To find blocks that contain relevant information for Austrian IPv6 infrastructure, the file can either be processed by using the Unix command `cat` combined with complex regular expressions or by parsing and storing the content of the file in a separate database and performing post processing.

The challenge of regular expressions is, that even if the defined expression handles the unequal number of attributes per block (e.g: it is possible to define more than one `descr` attribute per block), the output is once more a large file that is indeed trimmed, but nonetheless difficult to process and to interpret.

The second solution, the parsing and import into a database, makes the processing easier and more flexible. The only challenge is to find a suitable database and a working way to parse the large file. The conclusion is, that RIPE NCC itself does not provide any proper way to parse or convert their database for import into another database format and that there is no other way than writing an own parser. Fortunately the Github user

[4] https://www.ripe.net/publications/docs/ripe-358
[5] ftp://ftp.ripe.net/ripe/
[6] ftp://ftp.ripe.net/ripe/dbase/split/ripe.db.inet6num.gz

```
%Tags relating to '2001:628:14fe::/48'
%RIPE-USER-RESOURCE

inet6num:       2001:628:400::/48
netname:        TUNET-V6A
descr:          LAN Technical University of Vienna
country:        AT
admin-c:        DUMY-RIPE
tech-c:         DUMY-RIPE
status:         ALLOCATED-BY-LIR
mnt-by:         RIPE-NCC-HM-MNT
created:        2002-04-04T15:41:55Z
last-modified:  2005-05-18T12:31:06Z
source:         RIPE
remarks:        ****************************
remarks:        * THIS OBJECT IS MODIFIED
remarks:        * Please note that all data that is
remarks:        * generally regarded as personal
remarks:        * data has been removed from this object.
remarks:        * To view the original object, please query
remarks:        * the RIPE Database at:
remarks:        * http://www.ripe.net/whois
remarks:        ****************************
```

Table 5.3: IPv6 TUWIEN Block in ripe.db.inet6num

Christian Mehlmauer provides an import of the IPv4 split of the RIPE Database into a PostgreSQL[7] database and placed the source code into Gitbub repository for disposal [8]. The "Ripe Database Parser" is written in Python and uses the external libraries netaddr[9], psycopg2[10], SQLAlchemy [11] and wheel[12]. After installing these libraries, a PostgreSQL instance has to be set up. The connection string, containing the path of the SQL instance and the credentials, has to be placed in the Python file `helper.py`. Further adaptions have to be done to enable the Ripe Database Parser to parse IPv6 addresses instead of IPv4 addresses. This is easily done by replacing the string "inetnum" by "inet6num" in all files and by commenting the parts where the ranges of IPv4 addresses are calculated. By executing the parser, it creates a new database in the PostgreSQL instance called "ripe" and places a new table "block" in there. The table contains the following attributes: id,

[7] https://www.postgresql.org/
[8] https://github.com/FireFart/ripe
[9] https://pypi.python.org/pypi/netaddr
[10] https://pypi.python.org/pypi/psycopg2
[11] https://pypi.python.org/pypi/SQLAlchemy/1.0.13
[12] https://pypi.python.org/pypi/wheel

inet6num, netname, description, country, maintained by. For each block in the RIPE Database file, an entry is created in the PostgreSQL database. To get all entries that are located in Austria, a SQL command has to be executed (Table 5.4).

Command
`Select * from block where country = 'AT'`

Table 5.4: SQL Query all Austrian registered IPv6 networks

As written in the documentation of the RIPE Database, the attribute `country` is not reliable regarding its content. Before the decision which prefix is consulted for analyzing and scanning is made, the description attribute has to be read. Usually this attribute gives more information about the entry and the location of the network.

5.2 Recognizing Patterns

Compared to IPv4 scans, IPv6s scans need more preparation. Because the address length has been increased to 128 bit, it is not possible to scan an entire subnetwork by simply bruteforcing the target addresses. To find targets anyway, the target address space has to be analyzed in further detail. The draft of RFC 7707 deals with detecting host addresses, which means the focus is set on the host address space which has at least a length of 64 bits. The number of possible targets that can build within one network is approximately $1.844 * 10^{19}$[63]. Even ZMap, which supports configuration for sending packets with the Kernel driver `PF_RING` which allows to send approximately 15 million packets per second [43], would need approximately $1.4 * 10^{7}$ days to scan all possible addresses. This number demonstrates, how important it is to reduce the search space to find targets.

Summarized, there are mainly two steps to get target addresses. The first one, which is optional but makes it significantly easier to get results, is to collect existing target addresses within one network by searching public accessible databases like DNS zones. The second one is to recognize the address plan that was used to address the hosts. Because IPv6 addresses are not as easy to remember as IPv4 addresses, administrator often use simple patterns to address their hosts. The easiest one is the so called "Low-byte Address" pattern. By using this pattern, most bytes are set to 0 except the least significant byte, which will be incremented for each host. It is also very common to embed the existing IPv4 address into the IPv6 address (e.g. 2001:db8::192.0.2.1) [63].

According to RFC 7707, about 90% of the observed mail servers and 70% of the observed routers are using the Low-byte Address pattern for addressing their IPv6 interface. About 25% of the observed web servers are using embedded IPv4 address but only 6% are allocating random IPv6 addresses. Most random addresses are applied to clients (about 70%) [63], which could be a hint that clients are using mainly the recommended Stateless Address Auto Configuration (SLAAC) or a DHCPv6 server with a large IPv6 pool and

random distribution for addressing or both in combination where SLAAC is used for generating addresses and the DHCPv6 server for providing options [64].

The problem of the large address space also exists for scanning the routing infrastructure. It is indeed easier to reduce the routing prefix from 64 bits to 32 bits by consulting the databases of the RIR (e.g.: RIPE Database), but the problem in finding the host address of a router still remains. Therefore, to find router addresses and valid prefixes, the use of the tool Traceroute6 [13] is recommended [63]. Traceroute6 is the IPv6 compatible version the IPv4 tool "traceroute" and tracks the route to the host by sending ICMPv6 packets to the host address. The IPv6 header field `Hop Limit` of the first packet is set to 1 by default. This forces the first routing node to respond to the sending host with the ICMPv6 message "Time exceeded: Hop limit" [9]. The next packet increases the hop limit until the target host is reached [65]. The result is a list of routing points, that respond on the the way to the target host. This list can be used to find patterns in the IPv6 address allocation of routers which can be used as input for the high performance scanner.

5.3 Parameter Optimization of ZMapv6

To perform optimal IPv6 scans with ZMap, it is important to set up ZMap correctly. ZMap offers a large set of execution parameters that help to adjust each run explicitly and provide the information that is required. The most important parameters which were used within the observations of this master theses are listed and summarized here.

- `--bandwidth`: The bandwidth defines the sending rate of ZMap in "bits per second". Variations regarding the bandwidth result in different quality of the scanning output. Therefore it is important to choose the correct bandwidth for each scan. The first point that has to be considered is the bandwidth which the host of the scanner has in disposal (i.e. bandwidth of the hardware adapter and bandwidth provided by the ISP). If the value of the bandwidth parameter is too high, which means larger than the provided bandwidth, the packets will be discarded without any notification and will not be sent to the target host. The second consideration is the bandwidth of the target network. Because of the length of one subnetwork, it is common that only one subnetwork is focused during a scan. The challenge is to identify the maximum bandwidth of the target network and additionally not to harm the routing point of this target network by overloading it. Different findings of the several scans against the same network could be a hint, that the full bandwidth capacity is used. This could result in grave problems, because the implemented permutation of target addresses reveals not the desired effect of relieving target routing nodes. If scans are only focused on the routing infrastructure of an ISP, the occurrence of such problems can not be expected, because usually the resources of an ISP are much larger than the provided bandwidth for an end user subnetwork.

[13] `http://linux.die.net/man/8/traceroute6`

- `--sender-threads`: This parameter defines the number of threads that are used for the sending process. It allows to send several packets simultaneously. It is obvious, that the more threads for sending packets are defined, the faster ZMap is able to send packets within a specific time. The number of chosen threads must be related to the used CPU and the number of cores the CPU provides. It is important not to forget that at least one thread is required for receiving packets and that the sending process must not reserve all resources of the host machine.

- `--verbosity`: The verbosity expects a number between 1-5. If the verbosity level is set to the default value 1, only basic information about the performed scan is listed . If the verbosity level is set to 5, all logging messages, including debug messages are printed.

- `--summary` : gives a detailed summary about a performed scan.

- `--probe-module`: The used Probe Module is also defined by a parameter. This parameter expects the name of a probe as input. For the IPv6 scans of this master thesis, the specially developed probe `icmp6_echoscan` was defined.

- `--interface`: Specifies the hardware interface which is used for sending and receiving probes. This is especially important, if the host machine provides more than one interface. The available interfaces can be listed by executing the command `ifconfig` in the shell of the OS. The parameter expects the name of the interface.

- `--output-file`: Expects the path of the output file. The default output is a comma separated list containing the addresses of the hosts that are replying to the requests. The definition of the output-file is additional to the specification of the Output Module.

- `--dryrun`: If ZMap is run in Dryrun-mode, it actually will not send any packets. Rather the packets are forwarded and printed by the Output Module. It shows the content of the Ethernet header, IPv6 header and the header of the protocol of the Probe Module of each packet.

CHAPTER 6

Results

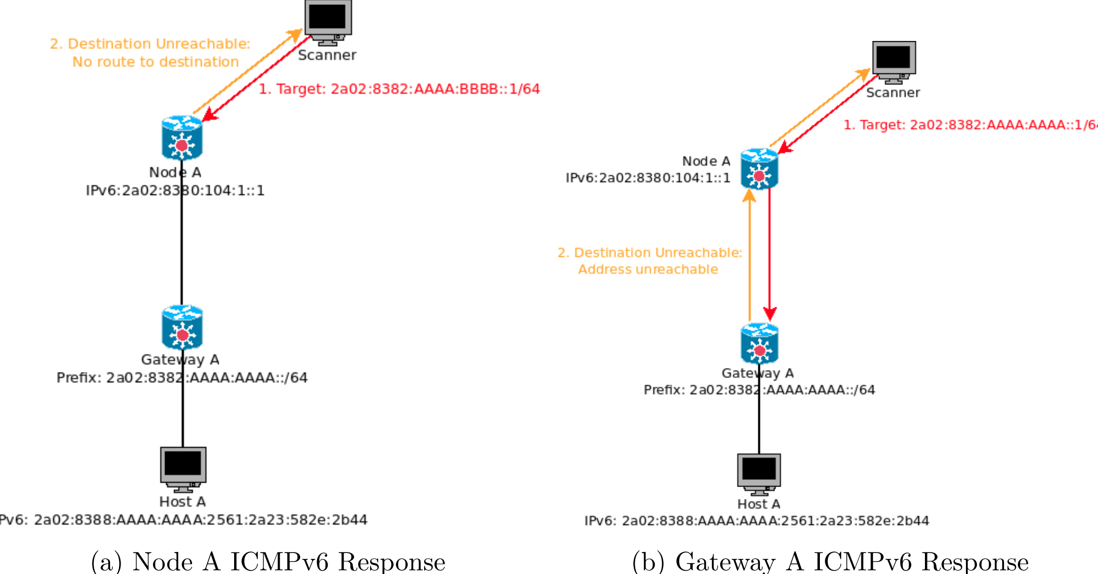
To obtain the following results, two computers in different geographical locations were used. This led to two different ISPs, two different IPv6 prefixes and therefore at least two different ways the packets were routed to their scanning targets. This setup was especially helpful when it came to prefix scanning of the Austrian ISP UPC Austria Services GmbH. For security purposes, some of the observed IPv6 addresses and IPv6 prefixes and the utilized IPv6 prefixes are masqueraded in this chapter. The results represent the proof of concept and the steps that where taken to come to the conclusions.

The chapter is divided into two parts. The first section deals with the performed prefix scanning, that tried to analyze the prefixes that are allocated by Austrian ISPs. The two different located computers led to two kind of observations (Observation A and Observation B). Because the results of both observations were different than expected, some tests were executed manually, without the high performance scanning approach (Test 1-5). The second section discusses the performed scans based on address pattern recognition, that used the masquerade permutation of the ZMap IPv6 extension. Additionally the results of these scans are grouped into the two categories found host nodes and found backbone nodes.

6.1 Prefix Scanning

The goal of the prefix scanning was to determine the amount of prefixes which are already routed by a well known Austrian ISP like UPC Austria Services GmbH. By performing this scans periodically, the distribution rate of IPv6 networks should be identified. Additionally, by identifying prefixes that are definitely in use, the search space of target hosts can be reduced significantly. At the moment it is hardly possible to find at least one random target address within a 64 bits prefix, that will respond to a probing packet. This would allow to draw a conclusion from used prefixes. Instead, the idea was, that a routing node, that is responsible for managing a prefix, will reply with an ICMPv6 message

of type 1 (Destination Unreachable Message 2.2.2) code 1 (No route to destination), if the target prefix is not available (see figure 6.1a). Furthermore, the last gateway of the routing trace will reply with an ICMPv6 message of type 1 and code 3 (Address unreachable) if the prefix will be routed, but the target address is not in use (see figure 6.1b).



(a) Node A ICMPv6 Response

(b) Gateway A ICMPv6 Response

Figure 6.1: Expected responses of ISP Prefix scanning.

The conclusion of the hypothesis is therefore, that the prefix will be routed and in use if:

- the randomly chosen target address actually belongs to a host and this host responds to a probe. This scenario is most unlikely but has to be considered.
- the responsible gateway responds with an ICMPv6 message, that indicates that the target address is not available.
- there is no response at all. This case should occur, if the incoming traffic is blocked by the gateway's firewall.

Otherwise, one gateway in the routing trace should respond to the request with a "No route to destination" reply. Therefore the packet capturing focus was set on reply packages of that type during a scan. This led to the evidently theory: The set of prefixes that were scanned, excluding the set of prefixes, that were part of a payload of a "No route to destination" message during the scan, leads to the wanted set of routed prefixes.

One of the used scanning machines was part of the network of the Austrian ISP UPC Austria Services GmbH. This means for the client there was an own 64 bit prefix within the ISP network available. This made it easy to determine other prefixes that are

distributed to other end customers by permuting the range between the prefix after the first 32 bits and the first 64 bits of the already known prefix. The ZMap extension for prefix permutation (see subsection 4.2.3 Prefix Permutation) was applied and the Probe Module for ICMPv6 Echo Request was loaded. The observed 32 bits target prefix was 2a02:8382::/32, which allowed the scanning of 2^{32} 64 bits prefixes (from 2a02:8382::/64 to 2a02:8382:FFFF:FFFF::/64).

The result of the scans differed, depending on which machine was used to perform the scans.

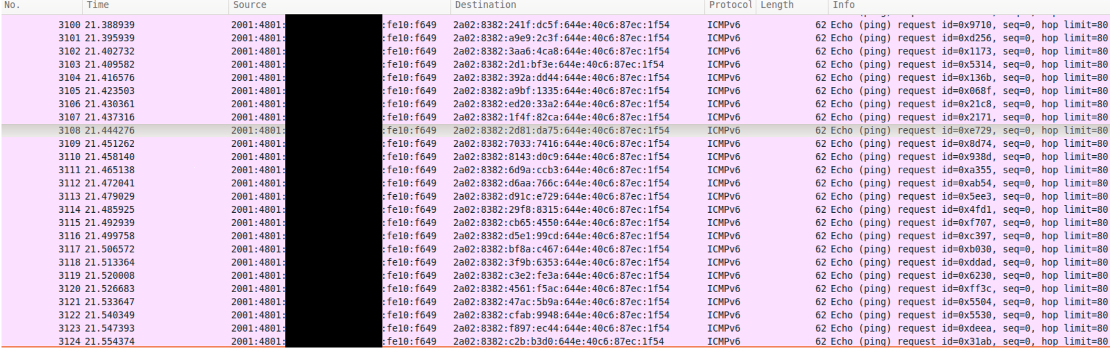
Observation A

Observation A describes the outcome of the scans with host A, that was outside the network of the ISP.

No node responded to any of the sent probing packets. Neither with message code 1 nor with code 3 of ICMPv6 message type 1. The Figure 6.2 shows some of the scanning probes, captured with the tool tcpdump[1]. The following filter was applied 6.1:

Command
This command captures the complete ICMPv6 traffic and stores it in the file dumpfile.pcap: `tcpdump -w dumpfile.pcap -nli eth0 icmp6`

Table 6.1: Capturing the ICMPv6 traffic with tcpdump



No.	Time	Source	Destination	Protocol	Length	Info
3100	21.388939	2001:4801: :fe10:f649	2a02:8382:241f:dc5f:644e:40c6:87ec:1f54	ICMPv6	62	Echo (ping) request id=0x9710, seq=0, hop limit=80
3101	21.395939	2001:4801: :fe10:f649	2a02:8382:a9e9:2c3f:644e:40c6:87ec:1f54	ICMPv6	62	Echo (ping) request id=0xd256, seq=0, hop limit=80
3102	21.402732	2001:4801: :fe10:f649	2a02:8382:3aa6:4ca8:644e:40c6:87ec:1f54	ICMPv6	62	Echo (ping) request id=0x1173, seq=0, hop limit=80
3103	21.409582	2001:4801: :fe10:f649	2a02:8382:2d1:bf3e:644e:40c6:87ec:1f54	ICMPv6	62	Echo (ping) request id=0x5314, seq=0, hop limit=80
3104	21.416576	2001:4801: :fe10:f649	2a02:8382:392a:dd44:644e:40c6:87ec:1f54	ICMPv6	62	Echo (ping) request id=0x136b, seq=0, hop limit=80
3105	21.423503	2001:4801: :fe10:f649	2a02:8382:a9bf:1335:644e:40c6:87ec:1f54	ICMPv6	62	Echo (ping) request id=0x068f, seq=0, hop limit=80
3106	21.430361	2001:4801: :fe10:f649	2a02:8382:ed20:33a2:644e:40c6:87ec:1f54	ICMPv6	62	Echo (ping) request id=0x21c8, seq=0, hop limit=80
3107	21.437316	2001:4801: :fe10:f649	2a02:8382:1f4f:82ca:644e:40c6:87ec:1f54	ICMPv6	62	Echo (ping) request id=0x2171, seq=0, hop limit=80
3108	21.444276	2001:4801: :fe10:f649	2a02:8382:2d81:da75:644e:40c6:87ec:1f54	ICMPv6	62	Echo (ping) request id=0xe729, seq=0, hop limit=80
3109	21.451262	2001:4801: :fe10:f649	2a02:8382:7033:7416:644e:40c6:87ec:1f54	ICMPv6	62	Echo (ping) request id=0x8d74, seq=0, hop limit=80
3110	21.458140	2001:4801: :fe10:f649	2a02:8382:8143:d0c9:644e:40c6:87ec:1f54	ICMPv6	62	Echo (ping) request id=0x938d, seq=0, hop limit=80
3111	21.465138	2001:4801: :fe10:f649	2a02:8382:6d9a:ccb3:644e:40c6:87ec:1f54	ICMPv6	62	Echo (ping) request id=0xa355, seq=0, hop limit=80
3112	21.472041	2001:4801: :fe10:f649	2a02:8382:d6aa:766c:644e:40c6:87ec:1f54	ICMPv6	62	Echo (ping) request id=0xab54, seq=0, hop limit=80
3113	21.479029	2001:4801: :fe10:f649	2a02:8382:d91c:e729:644e:40c6:87ec:1f54	ICMPv6	62	Echo (ping) request id=0x5ee3, seq=0, hop limit=80
3114	21.485925	2001:4801: :fe10:f649	2a02:8382:29f8:8315:644e:40c6:87ec:1f54	ICMPv6	62	Echo (ping) request id=0x4fd1, seq=0, hop limit=80
3115	21.492939	2001:4801: :fe10:f649	2a02:8382:cb65:4550:644e:40c6:87ec:1f54	ICMPv6	62	Echo (ping) request id=0xf707, seq=0, hop limit=80
3116	21.499758	2001:4801: :fe10:f649	2a02:8382:d5e1:99cd:644e:40c6:87ec:1f54	ICMPv6	62	Echo (ping) request id=0xc397, seq=0, hop limit=80
3117	21.506572	2001:4801: :fe10:f649	2a02:8382:bf8a:c467:644e:40c6:87ec:1f54	ICMPv6	62	Echo (ping) request id=0xb030, seq=0, hop limit=80
3118	21.513364	2001:4801: :fe10:f649	2a02:8382:3f9b:6353:644e:40c6:87ec:1f54	ICMPv6	62	Echo (ping) request id=0xddad, seq=0, hop limit=80
3119	21.520008	2001:4801: :fe10:f649	2a02:8382:c3e2:fe3a:644e:40c6:87ec:1f54	ICMPv6	62	Echo (ping) request id=0x6230, seq=0, hop limit=80
3120	21.526683	2001:4801: :fe10:f649	2a02:8382:4561:f5ac:644e:40c6:87ec:1f54	ICMPv6	62	Echo (ping) request id=0xff3c, seq=0, hop limit=80
3121	21.533647	2001:4801: :fe10:f649	2a02:8382:47ac:5b9a:644e:40c6:87ec:1f54	ICMPv6	62	Echo (ping) request id=0x5504, seq=0, hop limit=80
3122	21.540349	2001:4801: :fe10:f649	2a02:8382:cfab:9948:644e:40c6:87ec:1f54	ICMPv6	62	Echo (ping) request id=0x5530, seq=0, hop limit=80
3123	21.547393	2001:4801: :fe10:f649	2a02:8382:f897:ec44:644e:40c6:87ec:1f54	ICMPv6	62	Echo (ping) request id=0xdeea, seq=0, hop limit=80
3124	21.554374	2001:4801: :fe10:f649	2a02:8382:c2b:b3d0:644e:40c6:87ec:1f54	ICMPv6	62	Echo (ping) request id=0x31ab, seq=0, hop limit=80

Figure 6.2: Scanning from outside the ISP network.

On basis of the already mentioned criteria, the result of the observations would lead to the conclusion that every single scanned IPv6 prefix is routed and distributed. But this conclusion seems to be wrong. Because it would be very unlikely that the hit rate was that high, further observations were made manually by using the tools `tracert6` and `ping6`.

[1] http://www.tcpdump.org/tcpdump_man.html

Test 1
Because host B is part of the observed ISP network, first host A tried to "ping" the IPv6 address of host B to check if host A is even able to reach the target network. This test was positive and host B was able to respond to the ICMPv6 Echo Requests of host A.

Test 2
Host A tried to ping a random generated target address that is part of the known 64 bit prefix of host B. To perform the test correctly, it had to be ensured that this generated target address is not allocated by any host of the target network. The result of this test was no response by any node.

Test 3
Host A used `traceroute6` to discover the route to host B. The last node, that responded before host B was reached, is part of the ISPs routing infrastructure. This was verified by querying the RIPE NCC database (see section 5.1). The gateway of host B sent no response.

Test 4
Host A used `traceroute6` to discover the route to the random generated target address of test 2. The result was compared to the trace of test 3. Both traces were identical, except the fact, that in test 4 the tool ran into a timeout because no host listened to the target IPv6 address. Regarding the assumptions, the gateway should have sent an ICMPv6 message of type 1 code 3 (Address Unreachable).

Test 5
Host A used `traceroute6` to discover the route to a target prefix, which can be assumed not to be in use, but is regarding the RIPE NCC database (refTargetNetworks) related to the observed ISP (2a02:8388:FFFF:FFFF::1). This time, the trace stopped at a routing point that is not part of the observed ISP. But again, traceroute simply ran into a timeout and no ICMPv6 message of type 1 was sent by any node.

6.1.1 Conclusion of Observation A

The result of the scans and the subsequent, manually applied tests let assume, that the observed ISP disabled ICMPv6 error and informational messages. This makes it impossible to use the scanning approach to determine routed prefixes. The tests showed that the end customer's gateways do not respond to any ICMPv6 traffic that comes from an untrusted source. This refutes the assumption about the ICMPv6 behavior at least for this observed ISP and makes it therefore impossible to use high performance scanning from outside for analyzing end consumer prefixes.

6.1.2 Observation B

Observation B describes the outcome of the scans with host B, that was located inside the network of the observed ISP. Because the upload of the Internet connection of client B was limited by the ISP to 10Mbit/s , the scanning speed of ZMap had to be restricted.

Compared to scans of host A (which had a limit of 70Mbit/s), it would take much more time to perform a full scan with 32^{32} permutations. Therefore the scans had to be interrupted after some run time. This is not optimal, but will not distort the result and the conclusion of the observation.

Figure 6.3 shows the captured ICMPv6 traffic filtered by message type 1. At first view, it looks like the assumptions about prefix scanning are fulfilled. Actually one routing node of the observed ISP responded on not routable packets with ICMPv6 messages that indicate that there is no route to the target host. But the number of messages of type 1 is conspicuous. 4.431 ICMPv6 packets, including the outgoing Echo Request messages, were captured. Only 30 of them were of type "Desination Unreachable". After 20 repetitions, the results stayed almost the same to the first scan, which indicates that the result is not reliable. To prove this speculation, again manual tests were applied.

Filter: icmpv6.type==1 Expression... Clear Apply Save

No.	Time	Source	Destination	Protocol	Length	Info
111	0.751527	2a02:8380:104:1::1	2a02:8388: :2561:2a23:582e:2b44	ICMPv6	110	Destination Unreachable (no route to destination)
258	1.755494	2a02:8380:104:1::1	2a02:8388: :2561:2a23:582e:2b44	ICMPv6	110	Destination Unreachable (no route to destination)
405	2.760415	2a02:8380:104:1::1	2a02:8388: :2561:2a23:582e:2b44	ICMPv6	110	Destination Unreachable (no route to destination)
551	3.765279	2a02:8380:104:1::1	2a02:8388: :2561:2a23:582e:2b44	ICMPv6	110	Destination Unreachable (no route to destination)
698	4.766807	2a02:8380:104:1::1	2a02:8388: :2561:2a23:582e:2b44	ICMPv6	110	Destination Unreachable (no route to destination)
844	5.771472	2a02:8380:104:1::1	2a02:8388: :2561:2a23:582e:2b44	ICMPv6	110	Destination Unreachable (no route to destination)
992	6.771798	2a02:8380:104:1::1	2a02:8388: :2561:2a23:582e:2b44	ICMPv6	110	Destination Unreachable (no route to destination)
1139	7.776820	2a02:8380:104:1::1	2a02:8388: :2561:2a23:582e:2b44	ICMPv6	110	Destination Unreachable (no route to destination)
1285	8.776934	2a02:8380:104:1::1	2a02:8388: :2561:2a23:582e:2b44	ICMPv6	110	Destination Unreachable (no route to destination)
1431	9.776970	2a02:8380:104:1::1	2a02:8388: :2561:2a23:582e:2b44	ICMPv6	110	Destination Unreachable (no route to destination)
1578	10.781329	2a02:8380:104:1::1	2a02:8388: :2561:2a23:582e:2b44	ICMPv6	110	Destination Unreachable (no route to destination)
1725	11.787148	2a02:8380:104:1::1	2a02:8388: :2561:2a23:582e:2b44	ICMPv6	110	Destination Unreachable (no route to destination)
1871	12.791674	2a02:8380:104:1::1	2a02:8388: :2561:2a23:582e:2b44	ICMPv6	110	Destination Unreachable (no route to destination)
2018	13.797041	2a02:8380:104:1::1	2a02:8388: :2561:2a23:582e:2b44	ICMPv6	110	Destination Unreachable (no route to destination)
2165	14.801446	2a02:8380:104:1::1	2a02:8388: :2561:2a23:582e:2b44	ICMPv6	110	Destination Unreachable (no route to destination)
2311	15.801503	2a02:8380:104:1::1	2a02:8388: :2561:2a23:582e:2b44	ICMPv6	110	Destination Unreachable (no route to destination)
2456	16.798500	2a02:8380:104:1::1	2a02:8388: :2561:2a23:582e:2b44	ICMPv6	110	Destination Unreachable (no route to destination)
2603	17.801881	2a02:8380:104:1::1	2a02:8388: :2561:2a23:582e:2b44	ICMPv6	110	Destination Unreachable (no route to destination)
2750	18.811777	2a02:8380:104:1::1	2a02:8388: :2561:2a23:582e:2b44	ICMPv6	110	Destination Unreachable (no route to destination)
2897	19.815859	2a02:8380:104:1::1	2a02:8388: :2561:2a23:582e:2b44	ICMPv6	110	Destination Unreachable (no route to destination)
3042	20.812378	2a02:8380:104:1::1	2a02:8388: :2561:2a23:582e:2b44	ICMPv6	110	Destination Unreachable (no route to destination)
3189	21.821941	2a02:8380:104:1::1	2a02:8388: :2561:2a23:582e:2b44	ICMPv6	110	Destination Unreachable (no route to destination)
3335	22.821765	2a02:8380:104:1::1	2a02:8388: :2561:2a23:582e:2b44	ICMPv6	110	Destination Unreachable (no route to destination)
3482	23.827445	2a02:8380:104:1::1	2a02:8388: :2561:2a23:582e:2b44	ICMPv6	110	Destination Unreachable (no route to destination)
3628	24.827431	2a02:8380:104:1::1	2a02:8388: :2561:2a23:582e:2b44	ICMPv6	110	Destination Unreachable (no route to destination)

Figure 6.3: Scanning from inside the ISP network.

Test 1
Host B executed an endless ping to one target address, that was also a generated target address of the high performance scan. During the scan, the routing node replied the Echo Request with a "Desination Unreachable" ICMPv6 message. Figure 6.4 shows the result of the performed pinging. 20 packets were sent, but only 14 messages of type "Destination Unreachable" were received. The list and the ping statistics indicates, that the response of the routing node is not reliable, because there are too many timeouts.

Test 2
Host B executed an endless ping to one target address, that was also a generated target address of the high performance scan. During the performed scan no reply, that was related to that address, was received. Figure 6.5 shows the result of the ping. Unexpectedly Ping received, similar to test 1, a lot of "Destination Unreachable" messages. But again, the responses are not stable. The node replied only to 15 of 18 packets with "Destination Unreachable" messages.

```
christoph@Think: ~

christoph@Think:~$ ping6 2a02:8382:ef9a:9205:644e:40c6:87ec:1f54
PING 2a02:8382:ef9a:9205:644e:40c6:87ec:1f54(2a02:8382:ef9a:9205:644e:40c6:87ec:1f54) 56 data bytes
From 2a02:8380:104:1::1 icmp_seq=2 Destination unreachable: No route
From 2a02:8380:104:1::1 icmp_seq=3 Destination unreachable: No route
From 2a02:8380:104:1::1 icmp_seq=4 Destination unreachable: No route
From 2a02:8380:104:1::1 icmp_seq=5 Destination unreachable: No route
From 2a02:8380:104:1::1 icmp_seq=6 Destination unreachable: No route
From 2a02:8380:104:1::1 icmp_seq=7 Destination unreachable: No route
From 2a02:8380:104:1::1 icmp_seq=11 Destination unreachable: No route
From 2a02:8380:104:1::1 icmp_seq=12 Destination unreachable: No route
From 2a02:8380:104:1::1 icmp_seq=13 Destination unreachable: No route
From 2a02:8380:104:1::1 icmp_seq=14 Destination unreachable: No route
From 2a02:8380:104:1::1 icmp_seq=17 Destination unreachable: No route
From 2a02:8380:104:1::1 icmp_seq=18 Destination unreachable: No route
From 2a02:8380:104:1::1 icmp_seq=19 Destination unreachable: No route
From 2a02:8380:104:1::1 icmp_seq=20 Destination unreachable: No route
^C
--- 2a02:8382:ef9a:9205:644e:40c6:87ec:1f54 ping statistics ---
20 packets transmitted, 0 received, +14 errors, 100% packet loss, time 19049ms

christoph@Think:~$
```

Figure 6.4: Pinging a "Destination Unreachable" address

```
christoph@Think: ~

christoph@Think:~$ ping6 2a02:8382:d5cf:d8a3:644e:40c6:87ec:1f54
PING 2a02:8382:d5cf:d8a3:644e:40c6:87ec:1f54(2a02:8382:d5cf:d8a3:644e:40c6:87ec:1f54) 56 data bytes
From 2a02:8380:104:1::1 icmp_seq=2 Destination unreachable: No route
From 2a02:8380:104:1::1 icmp_seq=3 Destination unreachable: No route
From 2a02:8380:104:1::1 icmp_seq=6 Destination unreachable: No route
From 2a02:8380:104:1::1 icmp_seq=7 Destination unreachable: No route
From 2a02:8380:104:1::1 icmp_seq=8 Destination unreachable: No route
From 2a02:8380:104:1::1 icmp_seq=9 Destination unreachable: No route
From 2a02:8380:104:1::1 icmp_seq=10 Destination unreachable: No route
From 2a02:8380:104:1::1 icmp_seq=11 Destination unreachable: No route
From 2a02:8380:104:1::1 icmp_seq=12 Destination unreachable: No route
From 2a02:8380:104:1::1 icmp_seq=13 Destination unreachable: No route
From 2a02:8380:104:1::1 icmp_seq=14 Destination unreachable: No route
From 2a02:8380:104:1::1 icmp_seq=15 Destination unreachable: No route
From 2a02:8380:104:1::1 icmp_seq=16 Destination unreachable: No route
From 2a02:8380:104:1::1 icmp_seq=17 Destination unreachable: No route
From 2a02:8380:104:1::1 icmp_seq=18 Destination unreachable: No route
^C
--- 2a02:8382:d5cf:d8a3:644e:40c6:87ec:1f54 ping statistics ---
18 packets transmitted, 0 received, +15 errors, 100% packet loss, time 17023ms

christoph@Think:~$
```

Figure 6.5: Pinging an address with no response

6.1.3 Conclusion of Observation B

In contrast to observation A, in observation B, Host B received at least some of the expected ICMPv6 messages of type 1 ("Destination Unreachable"). But as proven, not every packet addressed to a non-routable IPv6 address is replied with an ICMPv6 message of that type. The reason for this behavior can be found in the RFC4443, the specification of ICMPv6: "A Destination Unreachable message **SHOULD** be generated by a router" [9]. Further, the RFC describes, a ICMPv6 message of type 1 **MUST NOT** be generated if a packet is dropped due to congestion [9]. Especially the second specification can occur

during high performance scans. This is the reason, why the response rate is that low. But even if the scanning rate is more restricted to avoid congestion, the results are not reliable, because the routing node does not guarantee a related packet generation. This is proven in test 1 and test 2.

6.2 Address Pattern Scanning

The target space was observed in further detail to detect patterns that were used for addressing hosts (see section 5.2. Recognizing Patterns). This approach was successful. The implemented extension of ZMap, 4.2.4 Masquerade Permutation, allows to specify the target space in most flexible way. This enables to define the observed and assumed address pattern as input parameter of ZMap and to craft only the target addresses that belong to the specified target address space.

Appendix A contains a selection of discovered target addresses that were found by performing address-pattern based scanning. The results can be grouped into two different categories of found nodes:

6.2.1 Host nodes

The discovered host nodes have a 64-bit prefix and and can be seen as end consumer nodes of a route. This gives the assumption that these hosts provide mostly other services than routing services. It can be also assumed that the discovered hosts are located behind a gateway of an ISP and a firewall (or are the gateway). Therefore, a limitation of the available bandwidth has to be considered. To get significant statements about the completeness of found IPv6 addresses within a 64-it prefix, the scanning bandwidth has to be chosen wisely. Otherwise the probing packets will be discarded by the ISP gateway or firewall and no responses can be received.

As example of found host nodes, Table A.2 of appendix A shows the IPv6 Test-Lab Infrastructure of the Andritz AG. The addresses were found by scanning the "Customer 1" network of another Austrian ISP T-Mobile Austria GmbH, which was found by querying the RIPE NCC database (see 5.1). All found addresses belong to different web servers that provide several test web services (see Figure 6.6.

The challenge of this observation was to find the correct bandwidth that led to the same result when repeating the experiment several times. This example and many others showed, that 100 kilobits per second is the optimal bandwidth to scan targets within a 64-bit end consumer prefix without any rejection of packets. The disadvantage of limited scanning speed is obviously the time it takes to scan a network completely. While the scanning of a permutation of 16 bits takes about nine minutes, the scan of a 17 bits permutation requires about two hours. With this speed, scanning the maximum number of targets, which is limited by ZMap to 2^{32}, would take several days. Furthermore, the gateway of the target network has to manage the whole incoming and outgoing scanning traffic while doing an end consumer prefix scan. This is different to Internet-Wide scans

Figure 6.6: Andritz AG IPv6 Testlab

over the globe. Permuting the target addresses comes with no benefit in this case. This could lead to a DoS attack or at least to congestion of the gateway. Because of unknown hardware and unknown configuration (we do not know the hardware limitations of the gateway), the risks is too high to perform such scans and is therefore not recommended.

6.2.2 Backbone nodes

The patterns of routing nodes were found by analyzing the trace of a route to a target host by using the tool traceroute. This is a quite simple way to recognize the patterns that were used to address the backbone routing nodes. Table A.1 of appendix A contains the addresses of the responding nodes of the the Vienna Internet Exchange (VIX) [2]. Table A.3 of appendix A contains a list of backbone router addresses of the Austrian ISP UPC Austria Services GmbH and Table A.4 and Table A.5 of appendix A show 255 IPv6 addresses of the Austrian Bundesrechenzentrum GmbH. All addresses listed in these tables were discovered while high performance scans. Collecting information and addresses about the target network is the first step of performing penetration tests. Therefore, all discovered addresses can be used for further tests.

Even if the result of the tests were successful, the performed experiments showed that this

[2] https://www.vix.at/vixhome.html

approach is not suitable for all routing infrastructures. For example, another Austrian ISP A1 Telekom Austria AG, has some defense mechanisms implemented, that prevent to expose their routing nodes by high performance scanning. No matter how often scans with an equal setup were executed, exactly three routing nodes replied to the probing packets withing the first 10 seconds of execution. Because of permutation, the order of scanned addresses was different for every execution, which led to three different nodes that replied. Changing the scanning setup by varying the bandwidth, had no influence on the test result. This is another example that shows why it is important to repeat the same tests for several times to come to correct conclusions.

CHAPTER 7

Conclusion and Future Work

In this master thesis, the IPv4 high performance scanner ZMap was extended to support IPv6 scans natively. The most important steps are documented and can be used for other projects which research IPv6 support for existing C programs.

The scanner was used to examine two different approaches of IPv6 high performance scanning. The intention of the first approach, the prefix scanning, was to find out how far the distribution of IPv6 in Austria has been proceeded. Unfortunately, different implementations of processing ICMPv6 messages by ISP routing nodes and their reactions on incoming ICMPv6 messages, makes IPv6 high performance scanning not suitable to give meaningful answers to this question. Querying the database of RIPE NCC, which is also described in this thesis, gives a much better overview of IPv6 distribution in Austria. But the results of the database queries are not hundred percent reliable, because the database contains IPv6 networks that are reserved by a company or an end customer whether they are actually in use or not. Additionally, there is no constraint about the correct definition of the geographical location of the reserved network.

The second approach, scanning of addresses based on pattern-recognition, had a more satisfying success rate. Many host nodes and backbone routing nodes were found. The chosen bandwidth for the network scan had high influence on the number of responding nodes and the quality of the scans. This makes it necessary to repeat the same scans several times to get a complete list of replying nodes and to find the optimal bandwidth for every single target network. In case of the Austrian ISP A1 Telekom Austria GmbH, varying the bandwidth while scanning the identified backbone addresses did not lead to success. Nevertheless this approach can be used to gather information for further penetration tests and it showed that the high amount of possible IPv6 addresses alone will not preserve from malicious attacks.

Compared to other state of the art network scanners, the extensions of ZMap developed in this master thesis allow completely new approaches for analyzing IPv6 networks and

the IPv6 backbone infrastructure. During working on this thesis, researchers from the Technical University of Munich published their own version IPv6-capable ZMAP [44]. However, our version provides more functionality and flexibility regarding the definition of target networks. Table 7.1 gives a comparison of known network scanners and adaptions of ZMap.

Scanner	High Performance Scanner	Scanning single IPv6 targets	Scanning IPv6 hitlist	Scanning IPv6 prefix ranges	Scanning IPv6 address patterns	IPv6 target permutation
ZMap (original version)	✓	-	-	-	-	-
MASSCAN	✓	-	-	-	-	-
NMAPv7	-	✓	✓	-	-	-
ZMap (TU Munich version)	✓	✓	✓	-	-	-
ZMap (our version)	✓	✓	✓	✓	✓	✓

Table 7.1: Network Scanners by comparison

The performed experiments as well as the adaptions of ZMap leave numerous possibilities for extension. Concerning the experiments, the target range can be extended to international IPv6 networks and not just limited to Austrian networks. Furthermore, already observed networks can be scanned towards more address patterns as described for example in RFC 7707 [63].

The implemented extension of ZMap supports only probes of type ICMPv6 Echo Request. To find more target hosts and gain more knowledge about the evolving IPv6 infrastructure, more probes of different types would support those observations. Furthermore, the existing ICMPv6 Echo Request probe can be extended by payload validation to process ICMPv6 replies that are different to the message type ICMPv6 Echo Response.

To improve the performance of scanning, Durumeric [43] has already shown that the original version of ZMap supports scanning by using the network interface driver `PF_Ring`. This allows scanning bandwidth up to 10Gbps. `PF_Ring` was also considered and implemented for the IPv6 extension. But due to missing hardware, the implementation has never been tested.

The outcome of the second scanning approach, the pattern-based scanning, can be used to continue the IPv6 scanning observations of the institute of Informatics of the Technical University of Munich [44]. The found addresses can be used to extend their hitlist and to scan the addresses over time.

APPENDIX A

Addresses found through address pattern-based scanning

2001:7f8:30:0:2:1:0:8928	2001:7f8:30:0:1:1:0:3330	2001:7f8:30:0:2:1:0:3320
2001:7f8:30:0:1:1:0:1257	2001:7f8:30:0:1:1:0:8412	2001:7f8:30:0:2:1:0:9002
2001:7f8:30:0:1:1:0:6720	2001:7f8:30:0:2:1:0:1853	2001:7f8:30:0:2:1:0:5578
2001:7f8:30:0:2:1:0:286	2001:7f8:30:0:2:1:0:8445	2001:7f8:30:0:2:1:0:6830
2001:7f8:30:0:2:1:0:8596	2001:7f8:30:0:2:1:0:6939	2001:7f8:30:0:2:1:0:8075
2001:7f8:30:0:2:1:0:8591	2001:7f8:30:0:2:1:0:8400	2001:7f8:30:0:2:1:0:1764
2001:7f8:30:0:1:1:0:8245	2001:7f8:30:0:1:1:0:8218	2001:7f8:30:0:2:1:0:9119
2001:7f8:30:0:1:1:0:8220	2001:7f8:30:0:2:1:0:5403	2001:7f8:30:0:1:1:0:2906
2001:7f8:30:0:1:1:0:8339	2001:7f8:30:0:2:1:0:5588	2001:7f8:30:0:1:1:0:3856
2001:7f8:30:0:2:1:0:8387	2001:7f8:30:0:2:1:0:251	2001:7f8:30:0:1:1:0:42
2001:7f8:30:0:2:1:0:3212	2001:7f8:30:0:1:1:0:1120	

Table A.1: 32 addresses of the Vienna Internet eXchange (VIX)

2001:9d0:10::240	2001:9d0:10::60	2001:9d0:10::2
2001:9d0:10::110	2001:9d0:10::bbbb	2001:9d0:10::200
2001:9d0:10::111	2001:9d0:10::51	2001:9d0:10::3
2001:9d0:10::1	2001:9d0:10::1280	

Table A.2: 11 addresses of the Andritz AG IPv6 Test-Lab

2001:730:2800::5474:e718	2001:730:2800::5474:e72a	2001:730:2800::5474:e70e
2001:730:2800::5474:e730	2001:730:2800::5474:e78b	2001:730:2800::5474:e705
2001:730:2800::5474:8067	2001:730:2800::5474:8043	2001:730:2800::5474:e706
2001:730:2800::5474:e70b	2001:730:2800::5474:e72d	2001:730:2800::5474:e712
2001:730:2800::5474:e70f	2001:730:2800::5474:e727	2001:730:2800::5474:e787
2001:730:2800::5474:e713	2001:730:2800::5474:e703	2001:730:2800::5474:e71b
2001:730:2800::5474:8070	2001:730:2800::5474:e729	2001:730:2800::5474:e782
2001:730:2800::5474:e71c	2001:730:2800::5474:e72f	2001:730:2800::5474:e795
2001:730:2800::5474:e783	2001:730:2800::5474:e71a	2001:730:2800::5474:e786
2001:730:2800::5474:e726	2001:730:2800::5474:8068	2001:730:2800::5474:e793
2001:730:2800::5474:e707	2001:730:2800::5474:e70a	2001:730:2800::5474:e78a
2001:730:2800::5474:e71e	2001:730:2800::5474:e784	2001:730:2800::5474:e714
2001:730:2800::5474:e788	2001:730:2800::5474:e796	2001:730:2800::5474:e780
2001:730:2800::5474:e728	2001:730:2800::5474:e781	2001:730:2800::5474:e72c
2001:730:2800::5474:8004	2001:730:2800::5474:e715	2001:730:2800::5474:e72b
2001:730:2800::5474:8044	2001:730:2800::5474:e71d	2001:730:2800::5474:e72e
2001:730:2800::5474:807a	2001:730:2800::5474:8071	2001:730:2800::5474:e731
2001:730:2800::5474:e732	2001:730:2800::5474:800b	2001:730:2800::5474:8096
2001:730:2800::5474:e719	2001:730:2800::5474:e78c	

Table A.3: 56 backbone IPv6 addresses of the Austrian Internet Service Provider UPC

2a01:190:1701::92	2a01:190:1701::44a	2a01:190:1701::4e2
2a01:190:1701::401	2a01:190:1701::3d2	2a01:190:1701::2d1
2a01:190:1701::35a	2a01:190:1701::4ca	2a01:190:1701::25a
2a01:190:1701::499	2a01:190:1701::169	2a01:190:1701::252
2a01:190:1701::259	2a01:190:1701::23a	2a01:190:1701::1e2
2a01:190:1701::a4	2a01:190:1701::1b1	2a01:190:1701::451
2a01:190:1701::441	2a01:190:1701::281	2a01:190:1701::4d9
2a01:190:1701::61	2a01:190:1701::369	2a01:190:1701::341
2a01:190:1701::3cb	2a01:190:1701::142	2a01:190:1701::289
2a01:190:1701::191	2a01:190:1701::331	2a01:190:1701::3d9
2a01:190:1701::c2	2a01:190:1701::141	2a01:190:1701::2c2
2a01:190:1701::72	2a01:190:1701::4d1	2a01:190:1701::2f9
2a01:190:1701::12	2a01:190:1701::211	2a01:190:1701::421
2a01:190:1701::3e9	2a01:190:1701::392	2a01:190:1701::d9
2a01:190:1701::93	2a01:190:1701::6	2a01:190:1701::469
2a01:190:1701::33a	2a01:190:1701::47a	2a01:190:1701::4aa
2a01:190:1701::292	2a01:190:1701::81	2a01:190:1701::431
2a01:190:1701::4b1	2a01:190:1701::102	2a01:190:1701::11
2a01:190:1701::471	2a01:190:1701::3aa	2a01:190:1701::2b2
2a01:190:1701::1b9	2a01:190:1701::69	2a01:190:1701::199
2a01:190:1701::261	2a01:190:1701::3f9	2a01:190:1701::389
2a01:190:1701::489	2a01:190:1701::372	2a01:190:1701::249
2a01:190:1701::2ca	2a01:190:1701::459	2a01:190:1701::19
2a01:190:1701::149	2a01:190:1701::4f2	2a01:190:1701::271
2a01:190:1701::39a	2a01:190:1701::3c1	2a01:190:1701::4a2
2a01:190:1701::3a2	2a01:190:1701::1a9	2a01:190:1701::311
2a01:190:1701::262	2a01:190:1701::461	2a01:190:1701::51
2a01:190:1701::282	2a01:190:1701::79	2a01:190:1701::19a
2a01:190:1701::212	2a01:190:1701::129	2a01:190:1701::4da
2a01:190:1701::52	2a01:190:1701::1f2	2a01:190:1701::361
2a01:190:1701::2d9	2a01:190:1701::4e1	2a01:190:1701::414
2a01:190:1701::d2	2a01:190:1701::3d1	2a01:190:1701::419
2a01:190:1701::371	2a01:190:1701::13a	2a01:190:1701::20a
2a01:190:1701::382	2a01:190:1701::4a9	2a01:190:1701::4fa
2a01:190:1701::ca	2a01:190:1701::161	2a01:190:1701::40a
2a01:190:1701::422	2a01:190:1701::329	2a01:190:1701::239
2a01:190:1701::2fa	2a01:190:1701::3a9	2a01:190:1701::f2
2a01:190:1701::aa	2a01:190:1701::3a1	2a01:190:1701::4b9
2a01:190:1701::4f9	2a01:190:1701::91	2a01:190:1701::f9
2a01:190:1701::462	2a01:190:1701::209	2a01:190:1701::16a
2a01:190:1701::d1	2a01:190:1701::3ca	2a01:190:1701::3f2
2a01:190:1701::2bb	2a01:190:1701::1ba	2a01:190:1701::3b2

Table A.4: (225 IPv6 addresses of the Bundesrechenzentrum GmbH (1/2)

2a01:190:1701::474	2a01:190:1701::27a	2a01:190:1701::f5
2a01:190:1701::472	2a01:190:1701::45a	2a01:190:1701::409
2a01:190:1701::2e9	2a01:190:1701::391	2a01:190:1701::491
2a01:190:1701::35b	2a01:190:1701::2b1	2a01:190:1701::2c1
2a01:190:1701::2c9	2a01:190:1701::101	2a01:190:1701::3fa
2a01:190:1701::6a	2a01:190:1701::a1	2a01:190:1701::192
2a01:190:1701::2f1	2a01:190:1701::4c2	2a01:190:1701::1d9
2a01:190:1701::f1	2a01:190:1701::309	2a01:190:1701::362
2a01:190:1701::29	2a01:190:1701::a3	2a01:190:1701::312
2a01:190:1701::f3	2a01:190:1701::1fa	2a01:190:1701::f4
2a01:190:1701::3f1	2a01:190:1701::12a	2a01:190:1701::4c9
2a01:190:1701::e9	2a01:190:1701::2f2	2a01:190:1701::a9
2a01:190:1701::89	2a01:190:1701::3da	2a01:190:1701::269
2a01:190:1701::2bc	2a01:190:1701::37a	2a01:190:1701::412
2a01:190:1701::399	2a01:190:1701::35d	2a01:190:1701::2e1
2a01:190:1701::9	2a01:190:1701::449	2a01:190:1701::4a1
2a01:190:1701::479	2a01:190:1701::379	2a01:190:1701::3e1
2a01:190:1701::492	2a01:190:1701::1a2	2a01:190:1701::2d2
2a01:190:1701::2ea	2a01:190:1701::40b	2a01:190:1701::3c9
2a01:190:1701::c9	2a01:190:1701::291	2a01:190:1701::ea
2a01:190:1701::3b1	2a01:190:1701::21	2a01:190:1701::251
2a01:190:1701::1a1	2a01:190:1701::413	2a01:190:1701::1d2
2a01:190:1701::229	2a01:190:1701::2a9	2a01:190:1701::4e9
2a01:190:1701::99	2a01:190:1701::381	2a01:190:1701::1
2a01:190:1701::48a	2a01:190:1701::1e1	2a01:190:1701::1f1
2a01:190:1701::24a	2a01:190:1701::2ba	2a01:190:1701::35c
2a01:190:1701::c1	2a01:190:1701::82	2a01:190:1701::4f1
2a01:190:1701::3e2	2a01:190:1701::26a	2a01:190:1701::1f9
2a01:190:1701::359	2a01:190:1701::4ea	2a01:190:1701::339
2a01:190:1701::ba	2a01:190:1701::1da	2a01:190:1701::411
2a01:190:1701::2b9	2a01:190:1701::4c1	2a01:190:1701::1b2
2a01:190:1701::1e9	2a01:190:1701::3ea	2a01:190:1701::a
2a01:190:1701::3c2	2a01:190:1701::1d1	2a01:190:1701::a2

Table A.5: 225 IPv6 addresses of the Bundesrechenzentrum GmbH (2/2)

List of Figures

2.1 IPv4 Header . . . 6
2.2 IPv6 Header . . . 10
2.3 ICMPv4 Header Echo Request/Reply . . . 16
2.4 ICMPv4 Header Destination Unreachable . . . 16
2.5 ICMPv6 Message Format . . . 17
2.6 ICMPv6 Header Echo Request/Reply . . . 18
2.7 ICMPv6 Header Destination Unreachable . . . 19
2.8 ICMPv6 Header Time Exceeded . . . 19
2.9 ICMPv6 Header Neighbor Solicitation . . . 20
2.10 ICMPv6 Header Neighbor Advertisement . . . 21
2.11 ICMPv6 Header Router Solicitation . . . 21
2.12 ICMPv6 Header Router Advertisement . . . 22

3.1 ZMap architecture [4] . . . 28
3.2 ZMap dependencies from technical point of view . . . 29

4.1 ZMap adaptions in technical point of view . . . 40

6.1 Expected responses of ISP Prefix scanning. . . . 64
6.2 Scanning from outside the ISP network. . . . 65
6.3 Scanning from inside the ISP network. . . . 67
6.4 Pinging a "Destination Unreachable" address . . . 68
6.5 Pinging an address with no response . . . 68
6.6 Andritz AG IPv6 Testlab . . . 70

List of Tables

2.1 IPv4 Network Classes . . . 8
2.2 IPv4 Classless Inter-Domain Routing . . . 8
2.3 IPv4 Private Addresses . . . 8
2.4 IPv6 Address Space Classification . . . 11
2.5 General Format of an IPv6 Global Unicast Address . . . 12
2.6 Specific Format of an IPv6 Global Unicast Address . . . 13
2.7 Comparison between IPv4 and IPv6 . . . 14

4.1 Generate configuration file for new execution parameters . . . 41
4.2 Verification steps for all entries of the list returned by the function getiffaddr. 42
4.3 Macros for determining the type of IPv6 address. . . . 42
4.4 IPv6 Address Structure with Variable Prefix Length . . . 45
4.5 IPv6 Address Permutation with 32-bit Prefix . . . 45
4.6 IPv6 Address Permutation with 48-bit Prefix . . . 46
4.7 Mask 16 Bits of the Second 32 Bits with Prefixmask . . . 47
4.8 Merge Random Value and Masked IPv6 Address . . . 47
4.9 IPv6 Address Masking with IPv6 Mask . . . 49
4.10 Inserting the Random Generated Integer Value . . . 49
4.11 Detailed View: Merge Inserting Mask and Masked IPv6 Address . . . 51

5.1 RIPE NCC "Whois" Result of A1 DNS Server . . . 56
5.2 Read content of ripe.db.inet6num . . . 57
5.3 IPv6 TUWIEN Block in ripe.db.inet6num . . . 58
5.4 SQL Query all Austrian registered IPv6 networks . . . 59

6.1 Capturing the ICMPv6 traffic with tcpdump . . . 65

7.1 Network Scanners by comparison . . . 74

A.1 32 addresses of the Vienna Internet eXchange (VIX) . . . 75
A.2 11 addresses of the Andritz AG IPv6 Test-Lab . . . 75
A.3 56 backbone IPv6 addresses of the Austrian Internet Service Provider UPC . 76
A.4 (225 IPv6 addresses of the Bundesrechenzentrum GmbH (1/2) . . . 77
A.5 225 IPv6 addresses of the Bundesrechenzentrum GmbH (2/2) . . . 78

List of Algorithms

4.1 Inserting Mask Creation 50

Acronyms

AFTR Address Family Transition Router. 23

API Application Programming Interface. 29, 32

ARP Address Resolution Protocol. 19, 20, 30

CIDR Classless Inter-Domain Routing. 7, 8, 11, 44, 45

CLI Command-line Interface. 28

CPU Central Processing Unit. 31, 61

CSV Comma-separated values. 37

DHCP Dynamic Host Configuration Protocol. 14, 21

DHCPv6 Dynamic Host Configuration Protocol, version 6. 14, 59, 60

DNS Domain Name Service. 2, 55, 56, 59

DoS Denial-of-Service. 25, 70

DS Dual Stack. 1

DS Lite Dual Stack Lite. 1

FTP File Transfer Protocol. 57

GAN Global Area Network. 42

GDB GNU Debugger. 27

GMP GNU Multiple Precision Arithmetic Library. 35

IANA Internet Assigned Numbers Authority. 1, 12, 13, 18

ICMP Internet Control Message Protocol. 35, 36

ICMPv6 Internet Control Message Protocol version 6. 39

ioctl input/output control. 30, 31, 41

ISP Internet Service Provider. 1, 12, 23, 55, 56, 60, 63–67, 69–71, 73

JSON JavaScript Object Notation. 37

MAC Media Access Control. 30

MTU Maximum Transmission Unit. 22

NA Neighbor Advertisement. 19, 20

NAT Network Address Translation. 2, 8, 9, 14, 23

NS Neighbor Solicitation. 19, 20

OS Operating System. 29, 30, 32, 61

OSI model Open Systems Interconnection Model. 5, 9

pcap Packet Capture. 29, 32, 33, 36

QoS Quality of Service. 6, 9, 10

RA Router Advertisement. 20, 21

RFC Request for Comment. 5, 6, 9–13, 15, 17–19, 42, 59, 68, 74

RIPE NCC Réseaux IP Européens Network Coordination Centre. 13, 55, 57

RIR Regional Internet Registry. 12, 60

RIRs Regional Internet Registries. 13

RS Router Solicitation. 20, 21

SIIT Stateless IP/ICMP Translator Algorithm. 24

SLAAC Stateless Address Auto Configuration. 59, 60

SLAAC Stateless Address Autoconfiguration. 20

UDP User Datagram Protocol. 29

VPN Virtual Private Network. 12

Bibliography

[1] Dan Lüdtke. *IPv6-Workshop: Eine praktische Einführung in das Internet-Protokoll der Zukunft*. CreateSpace, 2013.

[2] Benedikt Stockebrand. *IPv6 in Practice: A Unixer's Guide to the Next Generation*. Berlin: Springer Verlag, 2007.

[3] Silvia Hagen. *IPv6 Essentials: Integrating IPv6 into Your IPv4 Network*. Sebastopol: O'Reilly Media, 2014.

[4] Zakir Durumeric, Eric Wustrow, and Alex Halderman. Zmap: Fast internet-wide scanning and its security applications. In *the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 605–620, Washington, D.C.: USENIX, 2013.

[5] Livia Dandrea-Böhm. A1 bietet IPv6 Internetadressen für Großkunden an. `https://www.a1.net/newsroom/2013/06/a1-bietet-ipv6-internetadressen-fur-groskunden-an`. [Online; accessed 24-August-2015].

[6] Alain Durand, Ralph Droms, James Woodyatt, and Yiu Lee. Dual-Stack Lite Broadband Deployments Following IPv4 Exhaustion. Internet-draft, 2011. `http://www.ietf.org/internet-drafts/draft-ietf-softwire-dual-stack-lite-11.txt`.

[7] Stephen Deering and Robert Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, 1998. `http://www.rfc-editor.org/rfc/rfc2460.txt`.

[8] Zakir Durumeric, James Kasten, and David Adrian. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 475–488, New York: ACM, 2014.

[9] Alex Conta, Stephen Deering, and Mukesh Gupta. Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. RFC 4443, 2006. `http://www.rfc-editor.org/rfc/rfc4443.txt`.

[10] Peter Mandl, Andreas Bakomenko, and Johannes Weiss. *Grundkurs Datenkommunikation: TCP/IP-basierte Kommunikation: Grundlagen, Konzepte und Standards*. Wiesbaden: Vieweg+Teubner Verlag, 2010.

[11] Jon Postel. INTERNET PROTOCOL. RFC 791, 1981. https://tools.ietf.org/html/rfc791.

[12] Andreas Keller. *Breitbandkabel und Zugangsnetze: Technische Grundlagen und Standards*. Heideberg: Springer, 2011.

[13] Kristof Obermann and Martin Horneffer. *Datennetztechnologien für Next Generation Networks: Ethernet, IP, MPLS und andere*. Wiesbaden: Vieweg+Teubner, 2009.

[14] J. D. Wegner and Robert Rockell. *IP Addressing and Subnetting, Including IPv6*. Rockland: Syngress Media, 2000.

[15] Vince Fuller, Dr. Tony Li, Kannan Varadhan, and Jessica Yu. Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy. RFC 1519, 2013. https://www.ietf.org/rfc/rfc1519.txt.

[16] Kjeld Borch Egevang and Paul Francis. The IP Network Address Translator (NAT). RFC 1631, 1994. https://tools.ietf.org/html/rfc1631.

[17] Robert M. Hinden and Stephen E. Deering. IP Version 6 Addressing Architecture. RFC 4291, 2006. https://tools.ietf.org/html/rfc4291.

[18] Brian Haberman and Robert M. Hinden. Unique Local IPv6 Unicast Addresses. RFC 4193, 2013. https://www.ietf.org/rfc/rfc4193.txt.

[19] Robert M. Hinden, Stephen E. Deering, and Erik Nordmark. IPv6 Global Unicast Address Format. RFC 3587, 2003. https://tools.ietf.org/html/rfc3587.

[20] Internet Architecture Board and Internet Engineering Steering Group. IAB/IESG Recommendations on IPv6 Address Allocations to Sites. RFC 3177, 2001. https://tools.ietf.org/html/rfc3177.

[21] Geoff Huston and Dr. Thomas Narten. Ipv6 address assignment to end sites. RFC 6177, 2015. https://www.ietf.org/rfc/rfc6177.txt.

[22] Ralph Droms. Dynamic Host Configuration Protocol. RFC 2131, 1997. https://www.ietf.org/rfc/rfc2131.txt.

[23] Stuart Cheshire, Bernard Aboba, and Erik Guttman. Dynamic Configuration of IPv4 Link-Local Addresses. RFC 3927, 2005. https://tools.ietf.org/html/rfc3927.

[24] Jim Bound, Bernie Volz, Ted Lemon, and Mike Carney. Dynamic Host Configuration Protocol for IPv6 (DHCPv6). RFC 3315, 2003. https://tools.ietf.org/html/rfc3315.

[25] Thomas Narten, Erik Nordmark, William Allen Simpson, and Hesham Soliman. IPv6 Stateless Address Autoconfiguration. RFC 4861, 2007. https://tools.ietf.org/html/rfc4862.

[26] Asjad Amin, Waqas Anjum, Muhammad Salman Malik, and Syed Noman Ali. Performance eveluation of ipv4 and ipv6 networks in absence of link layer protection. In *Electronics, Communications and Photonics Conference (SIECPC)*, pages 1–5, KACST Headquarters: IEEE, 2011.

[27] Margaret Wasserman and Fred Baker. IPv6-to-IPv6 Network Prefix Translation. RFC 6296, 2011. `https://tools.ietf.org/html/rfc6296`.

[28] Scott Hogg and Eric Vyncke. *IPv6 Security: Information assurance for the next-generation Internet Protocol*. Indianapolis: Cisco Press, 2009.

[29] J. Postel. INTERNET CONTROL MESSAGE PROTOCOL. RFC 792, 1981. `https://tools.ietf.org/html/rfc792`.

[30] *ping(8) - Linux man page*. `http://linux.die.net/man/8/ping`.

[31] Susan Thomson, Thomas Narten, and Tatuya Jinmei. Neighbor Discovery for IP version 6 (IPv6). RFC 4862, 2007. `https://tools.ietf.org/html/rfc4861`.

[32] I-Ping Hsieh and Shang-Juh Kao. Managing the co-existing network of ipv6 and ipv4 under various transition mechanisms. In *Third International Conference on Information Technology and Applications (ICITA'05) (Volume:2)*, pages 765–771, KACST Headquarters: IEEE, 2005.

[33] Tomasz Bilski. From ipv4 to ipv6 – data security in the transition phase. In *The Seventh International Conference on Networking and Services*, pages 66–72, Venices: ICNS, 2011.

[34] Robert E. Gilligan and Erik Nordmark. Basic transition mechanisms for ipv6 hosts and routers. RFC 4213, 2005. `https://www.ietf.org/rfc/rfc4213.txt`.

[35] Brian E. Carpenter and Keith Moore. Connection of ipv6 domains via ipv4 clouds. RFC 3056, 2001. `https://www.ietf.org/rfc/rfc3056.txt`.

[36] Dave Thaler Fred L. Templin, Tim Gleeson. Intra-site automatic tunnel addressing protocol (isatap). RFC 5214, 2008. `https://www.ietf.org/rfc/rfc5214.txt`.

[37] Christian Huitema. Teredo: Tunneling ipv6 over udp through network address translations (nats). RFC 4380, 2006. `https://www.ietf.org/rfc/rfc4380.txt`.

[38] Xing Li, Fred Baker, Kevin Yin, and Congxiao Bao. Framework for ipv4/ipv6 translation. RFC 6144, 2015. `https://www.ietf.org/rfc/rfc6144.txt`.

[39] Mark Osborne. *How to Cheat at Managing Information Security*. Rockland: Syngress Publishing, Inc., 2006.

[40] Andrew Whitaker and Daniel P. Newman. *Penetration Testing and Network Defense*. Indianapolis: Cisco Press, 2005.

[41] Paulino Calderón Pale. *NMAP6: Network exploration and security auditing Cookbook*. Birmingham: Packt, 2012.

[42] Gordon Lyon. Nmap 6 Release Notes. `https://nmap.org/6/`. [Online; accessed 01-March-2015].

[43] David Adrian, Zakir Durumeric, Gulshan Singh, and J. Alex Halderman. Zippier zmap: Internet-wide scanning at 10 gbps. In *8th USENIX Workshop on Offensive Technologies (WOOT 14)*, San Diego: USENIX Association, 2014.

[44] Oliver Gassera, Quirin Scheitle, Sebastian Gebhard, and Georg Carle. Scanning the ipv6 internet: Towards a comprehensive hitlist. In *Traffic Monitoring and Analysis - 8th International Workshop*, Louvain La Neuve, Belgium, 2016.

[45] Robert David Graham. MASSCAN: Mass IP port scanner. `https://github.com/robertdavidgraham/masscan`. [Online; accessed 25-February-2015].

[46] Peter Liggesmeyer. *Software-Qualtität: Testen, Analysieren und Verifizieren von Software*. Waltham: Elsevier, 2009.

[47] Robert Oshana and Mark Kraeling. *Software Engineering for Embedded Systems: Methods, Practical Techniques, and Applications*. Heideberg: Springer, 2013.

[48] Lorenzo Bettini. GNU Gengetopt 2.22.6. `https://www.gnu.org/software/gengetopt/gengetopt.html`. [Online; accessed 11-April-2016].

[49] Zhou Li, Tan Fang-Yong, and Gao Xiao-Hui. Research on application of netlink socket communication in linux ipsec support mechanism. In *the 5th International Conference on Computer Science & Education*, pages 933–936, Hefei: ICSSE, 2010.

[50] Maria Luisa Merani, Maurizio Casoni, and Walter Cerroni. *Hands-On Networking: From Theory to Practice*. Cambridge: Cambridge University Press, 2009.

[51] Luca Deri. Improving passive packet capture: Beyond device polling. In *4th International System Administration and Network Engineering Conference*, Amsterdam: SANE, 2004.

[52] Ntop.org. PF_RING™. `http://www.ntop.org/products/packet-capture/pf_ring/`. [Online; accessed 01-Mai-2016].

[53] Adrian David. 10GigE (Zippier) ZMap. `https://github.com/zmap/zmap/blob/master/10gigE.md`. [Online; accessed 02-Mai-2016].

[54] Liu Tianhua, Zhu Hongfeng, Liu Jie, and Zhou Chuansheng. The design and implementation of zero-copy for linux. In *Eighth International Conference on Intelligent Systems Design and Applications*, pages 121–126, Kaohsiung: IEEE, 2008.

[55] Joyce K. Reynolds and Jon Postel. ASSIGNED NUMBERS. RFC 1700, 1994. https://tools.ietf.org/html/rfc1700.

[56] *GETIFADDRS(3) Linux Programmer's Manual*, July 2015. http://man7.org/linux/man-pages/man3/getifaddrs.3.html.

[57] *ipv6 - Linux IPv6 protocol implementation; Linux Programmer's Manual*, March 2015. http://man7.org/linux/man-pages/man7/ipv6.7.html.

[58] Christian Huitema and Brian E. Carpenter. Deprecating site local addresses. RFC 3879, 2013. https://www.ietf.org/rfc/rfc3879.txt.

[59] Ranjit Bhatta and Sulav Malla. Bitwise Operators in C programming. http://www.programiz.com/c-programming/bitwise-operators. [Online; accessed 13-June-2016].

[60] Michael Beeler, R. William Gosper, and Richard Schroeppe. HAKMEM. Memo.

[61] Pili Hu. Bit Counting (Number of one's in an integer). http://blog.hupili.net/p--20130328-bit-counting/. [Online; accessed 01-April-2016].

[62] Réseaux IP Européens Network Coordination Centre (RIPE NCC).

[63] Fernando Gont and Tim Chown. Network Reconnaissance in IPv6 Networks. RFC 7707, 2016. https://tools.ietf.org/html/rfc7707.

[64] Tom Coffeen. *IPv6 Address Planning: Designing an Address Plan for the Future*. Sebastopol: O'Reilly Media, 2014.

[65] *traceroute6(8) - Linux man page*. http://linux.die.net/man/8/traceroute6.