DEGREE PROJECT IN INFORMATION AND COMMUNICATION TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, *SWEDEN 2020*

Evaluation of Network-Layer Security Technologies for Cloud Platforms

BRUNO MARCEL DUARTE COSCIA

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Evaluation of Network-Layer Security Technologies for Cloud Platforms

BRUNO MARCEL DUARTE COSCIA

Master in Security and Cloud Computing SECCLO
Date: 22.11.2020
Industry Supervisor: Bilal Ahmad, MSc.
KTH Supervisor: Hongyu Jin, PhD.
KTH Examiner: Professor Panagiotis Papadimitratos
Aalto University Examiner: Professor Tuomas Aura
School of Electrical Engineering and Computer Science
Host company: Oy LM Ericsson Ab (LMF), Finland
Swedish title: Utvärdering av säkerhetsteknologier för nätverksskiktet i molnplattformar

Abstract

With the emergence of cloud-native applications, the need to secure networks and services creates new requirements concerning automation, manageability, and scalability across data centers. Several solutions have been developed to overcome the limitations of the conventional and well established IPsec suite as a secure tunneling solution. One strategy to meet these new requirements has been the design of software-based overlay networks. In this thesis, we assess the deployment of a traditional IPsec VPN solution against a new secure overlay mesh network called Nebula. We conduct a case study by provisioning an experimental system to evaluate Nebula in four key areas: reliability, security, manageability, and performance. We discuss the strengths of Nebula and its limitations for securing inter-service communication in distributed cloud applications. In terms of reliability, the thesis shows that Nebula falls short to meet its own goals of achieving host-to-host connectivity when attempting to traverse specific firewalls and NATs. With respect to security, Nebula provides certificate-based authentication and uses current and fast cryptographic algorithms and protocols from the Noise framework. Regarding manageability, Nebula is a modern solution with a loosely coupled design that allows scalability with cloud-ready features and easier deployment than IPsec. Finally, the performance of Nebula clearly shows an overhead for being a user-space software application. However, the overhead can be considered acceptable in certain server-to-server microservice interactions and is a fair trade-off for its ease of management in comparison to IPsec.

Keywords

Overlay network, Network security, IPsec, Slack nebula, Nebula, Noise framework, Noise protocol

Sammanfattning

Med framväxten av molninbyggda applikationer skapar behovet av säkra nätverk och tjänster nya krav på automatisering, hanterbarhet och skalbarhet över datacenter. Flera lösningar har utvecklats för att övervinna begränsningarna i den konventionella och väletablerade IPsec-sviten som en säker tunnellösning. En strategi för att möta dessa nya krav har varit utformningen av mjukvarubaserade överläggsnätverk. I den här avhandlingen bedömer vi implementeringen av en traditionell IPsec VPN-lösning mot ett nytt säkert överläggsmeshnätverk som kallas Nebula. Vi genomför en fallstudie genom att bygga upp ett ett experimentellt system för att utvärdera Nebula inom fyra nyckelområden: tillförlitlighet, säkerhet, hanterbarhet och prestanda. Vi diskuterar styrkan i Nebula och dess begränsningar för att säkra kommunikation mellan tjänster i distribuerade molnapplikationer. När det gäller tillförlitlighet visar avhandlingen att Nebula inte uppfyller sina egna mål om att uppnå värd-till-värd-anslutning när man försöker korsa specifika brandväggar och NAT. När det gäller säkerhet tillhandahåller Nebula certifikatbaserad autentisering och använder aktuella och snabba kryptografiska algoritmer och protokoll från Noise-ramverket. När det gäller hanterbarhet är Nebula en modern lösning med en löst kopplad design som möjliggör skalbarhet med molnklara funktioner och enklare distribution än IPsec. Slutligen visar prestandan hos Nebula tydligt en overhead för att vara en användarutrymme-programvara. Dock kan kostnaderna anses vara acceptabla i vissa server-till-server-mikroserviceinteraktioner och är en rättvis avvägning om vi tar i betraktande dess enkla hantering jämfört med IPsec.

Nyckelord

Överlagring Nätverk, Nätverkssäkerhet, IPsec, Slack nebula, Nebula, Noise ramverk, Noise protokoll

Acknowledgements

I want to express my gratitude to the people who assisted and supported me in this project.

First, my gratitude goes to my thesis supervisors from Aalto University and KTH Royal Institute of Technology: Prof. Tuomas Aura and Prof. Panagiotis Papadimitratos for steering me in the right direction and their willingness to impart their knowledge.

I would also like to acknowledge the support of the advisors: MSc. Bilal Ahmad, and Ph.D. Hongyu Jin who gave valuable input in the thesis work.

This research would not have been possible without the backing of Ericsson Finland, specifically Ilkka Koskinen and the feedback of Andon Nikolov and Maarit Hietalahti.

My appreciation also goes to the SECCLO program staff for their trust in me to undertake this higher education and for enabling me to pursue my goals.

Finally, I must express my very profound gratitude to my family and all the other people that enabled me to be at this point in life and contribute to who I am. This accomplishment would not have been possible without them.

Thank you.

Otaniemi, 22.11.2020

Bruno Marcel Duarte Coscia

Contents

Symbols and abbreviations

Symbols

$\varnothing$	empty set for a not initialized variable
->	transmission of message from Alice to Bob
<-	transmission of message from Bob to Alice

Operators

$\leftarrow$	assignment operator
$\|\|$	concatenation operator

Abbreviations

ACL	Access Control List
AE	Authenticated encryption
AEAD	Authenticated encryption with associated data
AES	Advanced Encryption Standard Cipher Algorithm
AH	Authentication Header
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASN.1	Abstract Syntax Notation One
BLAKE2s	Cryptographic hash function successor of BLAKE-256
CA	Certificate authority
CBC	Cipher Block Chaining
CGN	Carrier-Grade NAT
CNA	Cloud-Native Applications
CRL	Certificate Revocation List
CSV	Comma-separated values
ChaCha20	Stream cipher related to ChaCha cipher

DH	Diffie–Hellman key exchange
DNS	Domain Name System
DevOps	Set of practices that combines Software Development (Dev) and Information-Technology Operations (Ops)
DoS	Denial-of-Service
EAP	Extensible Authentication Protocol
EIM	Endpoint-Independent Mapping
ESP	Encapsulated Security Protocol
FQDN	Fully Qualified Domain Name
GCM	Galois/Counter Mode
HKDF	HMAC-based Extract-and-Expand Key Derivation Function
HMAC	Keyed-Hashing for Message Authentication
HTTP	Hypertext Transfer Protocol
IANA	Internet Assigned Numbers Authority
ICE	Interactive Connectivity Establishment
ICMP	Internet Control Message Protocol
ICV	Integrity check value
IETF	Internet Engineering Task Force
IKEv2	Internet Key Exchange Protocol Version 2
IP	Internet Protocol
IPsec	Internet Protocol Security
IPv6	Internet Protocol version 6
ISP	Internet Service Provider
KCI	Key Compromise Impersonation
MTU	Maximum Transmission Unit
NAPT	Network Address and Port Translation
NAT	Network Address Translation
NGFW	Next Generation Firewall
NIST	National Institute of Standards and Technology
OAuth2	Open standard for access delegation version 2
OCSP	Online Certificate Status Protocol
OIDC	OpenID Connect
P2P	Peer-to-peer
PAD	Peer Authorization Database
PEM	Privacy-enhanced Electronic Mail
PKI	Public Key Infrastructure
Poly1305	Poly1305 is a cryptographic Message Authentication Code
QoS	Quality of Service
RFC	Request for Comments

RSA	Rivest-Shamir-Adleman cryptosystem for public-key encryption
SA	Security Association
SAD	Security Association Database
SAML	Security Assertion Markup Language
SCP	Secure Copy Protocol
SDN	Software-defined Networking
SIP	Session Initiation Protocol
SPD	Security Policy Database
SPI	Security Parameter Index
SSO	Single Sign-On
STUN	Session Traversal Utilities for NAT
TCP	Transmission Control Protocol
TCP/IP	Internet protocol suite commonly known as TCP/IP because the foundational protocols in the suite are the the Transmission Control Protocol (TCP) and the Internet Protocol (IP)
TLS	Transport Layer Security
TURN	Traversal Using Relays around NAT
UDP	User Datagram Protocol
VPN	Virtual Private Network
X.509	Cryptographic standard format for public key certificates
txqueuelen	Transmit Queue Length

Chapter 1

Introduction

The adoption of cloud computing has brought some challenges in securing communication between services. Previously, services that needed to be accessed from remote geographical sites or distinct branches of an organization used VPNs to extend and secure their network. Now, the services are intrinsically distributed and decentralized in multiple locations due to the ubiquity of cloud computing. Typically, VPN solutions have a centralized star topology or a mesh design. These solutions do not scale to cloud environments and would make the administration cumbersome. The default choice for creating secure VPNs has been IPsec for site-to-site use cases.

An attempt to overcome these limitations are so-called overlay networks [1]. Overlay networks are built on top of existing networks and offer a level of abstraction to enable scalability and easier administration, as in software-defined networks (SDN).

Nowadays, a similar concept is offered by so-called *service mesh* solutions that have been popularized to face the challenges of secure communication in cloud infrastructure. While software-defined networking is a more generic approach to network management, service meshes focus on the capabilities and the interaction of services in cloud-native applications. A service mesh brings an abstraction layer to secure the communication between microservices transparently without affecting the applications. However, current service mesh solutions are bound to an individual cloud provider or a specific cluster of virtual machines. In case an application is distributed across different cloud providers, unfortunately, the existing service mesh solutions cannot provide a unified global solution.

A new secure overlay network called Nebula claims to be a cross-cloud, cross-region, and cross-platform solution to fill this gap. This thesis studies

Nebula as a potential replacement for IPsec VPN to secure distributed applications, expecting better manageability without compromising security or performance.

1.1 Description of the problem

In organizations with distinct sites, a goal is for all communication to resemble the behaviour of local networks even when connecting geographically distant locations. Moreover, secure communication is crucial to protect the data traffic since it includes sensitive business and user information.

1.1.1 Geographically distributed services

IPsec with Encapsulated Security Protocol (ESP) as a session protocol combined with the network tunneling mode is known as IPsec VPN. Currently, it is the main choice to achieve secure communication between sites as shown in Figure 1.1.

The current solution composes the following setting:

- There are dedicated hardware devices with firewall capabilities placed on the border of each site facing the public internet.
- There is a proprietary IPsec VPN implementation, with partial hardware acceleration, from the firewall manufacturer.
- Static routes are configured on each firewall to enforce the IPsec policy on the inbound and outbound traffic that extends the local network behavior across sites.

Figure 1.1: VPN IPsec between two geographically distant locations.

1.1.2 Multiple sites

Eventually, the need to add more sites to the schema brings a set of complications listed below and illustrated in Figure 1.2.

- Another firewall with the same capabilities needs to be added to the new site.
- More static routes need to be added to each firewall to keep the local network behavior. This makes the solution not scalable.
- Inevitably, this leads to more IPsec VPN connections between the sites.
- In case of static routing, if one link is down or unavailable, there is no recovery or alternative rerouting of traffic for that link. This, however, does not apply when the nodes are connected through dynamic routing.

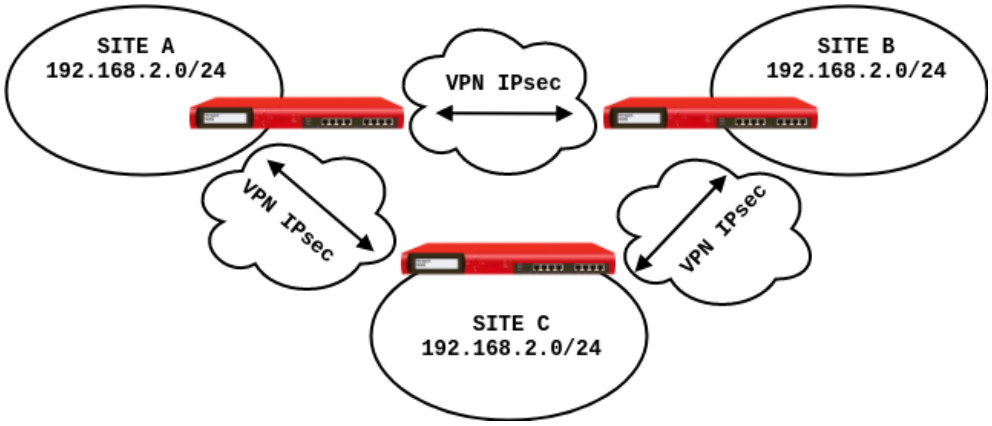

Figure 1.2: VPN IPsec between distant locations.

1.2 Research question

The conducted work compares Nebula and IPsec VPN.

We ask the following research question:

- Is Nebula a better solution than VPN IPsec to secure distributed applications that can bring ease of management without compromising security and performance?

1.3 Goal and objectives

The primary goal of this research work is to deduce if Nebula can replace IPsec VPN for extending local networks across geographically distant locations. The main objectives of this work can be summarized as follows:

- Examine the documentation, behavior, and the source code of Nebula and understand how it achieves the claims of global communication and confidentiality.
- Implement an experimental system to test secure communication solutions, including Nebula and IPsec VPN.

1.4 Methodology

This work falls under the **evaluation** category because of the emphasis of comparison, analysis, and assessment. The project combines the research method of *experimentation* and *case study* to achieve its objectives [2][3].

As an experiment:

- We define a concise hypothesis that the experiment will confirm or deny.
- We design and implement a provisioned experimental system to conduct the assessment.
- We control the variables of the experimental system.
- We analyze the performance of the technology.
- We report the procedures and results.

As a case study:

- We undertake a comprehensive exploration of the technology in the hypothesis and conclude the suitability of the solution based on qualitative and quantitative results.

1.5 Ethics and sustainability

The general purpose of this work is to evaluate secure overlay networks, specifically Nebula, in comparison with IPsec. Secure technology, such as Nebula aims to reduce crime and enable business.

We carry this evaluation using the open-source software repository provided by Slack, the company that developed Nebula. Also, we analyze the specification of the publicly available Noise framework, on which Nebula bases its cryptographic handshake for the communication. The work supports the adoption of open-source software for the benefit of everyone. The testbed configuration has been published as an open-source project under the MIT License [4].

On the sustainable aspect, we conducted the project during a pandemic, and the use of a virtual environment instead of physical labs enabled safe experiments. In addition, the virtualized environments favor the reduction of energy use and computational resources.

1.6 Scope and delimitation

We examine how Nebula achieves its connectivity compared to IPsec VPN. Therefore, some assumptions should be made:

- When connecting remote locations over the Internet, it is not possible to have full control over the network topology, or specifically over NATs and firewalls which the tunneled traffic needs to traverse.
- Nebula uses certificates for authenticating nodes; therefore, the rotation, blacklisting, and distribution of certificates its assume to follow the best practices, and it is out of scope in the present work.
- Companies commonly deploy IPsec VPN as a site-to-site solution because of the complications of host-to-host communication. However, the design of Nebula by itself is a host-to-host solution. System engineers willing to deploy Nebula should embrace this shift of paradigm and facilitate changes in the infrastructure to enable secure communication.

1.7 Structure of the report

We organize the thesis with the following structure. Chapter 2 provides a review of the literature and background for the project. Chapter 3 details the operation and architecture of Nebula and reviews the handshake for the key establishment in Nebula. Chapter 4 describes the testbed setup to conduct the experiments. Chapter 5 evaluates Nebula in four key areas: reliability,

security, manageability, and performance. Finally, Chapter 6 concludes the assessment and discusses future work.

Chapter 2

Background

This chapter summarizes some underlying technologies that enable secure communications such as IPsec and Nebula. It begins with a study of NAT, their classification, and why it represents a challenge when attempting host-to-host connectivity. Then, we review overlay networks and related existing technologies for secure communication. Last but not least, we present the overall operation and architecture of IPsec.

2.1 Network Address Translation (NAT)

Network Address Translation (NAT) refers to a procedure rather than a structured protocol, and it is usually present in routers and firewalls [5]. The procedure is applied to IP packets in transit. NAT allows hosts in a local and often private network to reach hosts in external and often public networks. To deliver the expected behavior, it comprises two operations that together are referred to as traditional NAT [6]:

1. Basic Network Address Translation or Basic NAT

   This operation changes the address space of an IP packet to another address space by seamlessly modifying the header of the IP packet without being noticeable by the end-user.

2. Network Address Port Translation or NAPT

   Network Address Port Translation (NAPT) is a method that groups many network addresses with their ports to a single IP address and its ports.

We show an example of the common procedure of the traditional NAT in Figure 2.1. Within the local area network, we identify devices (e.g., work-

stations, laptops, printers) with a Private-Use Network Address space of the type 192.168.100.0/24 [7]. Here, a computer identified by the local IP address 192.168.100.39/24 runs a web browser and attempts to access a remote web server identified by the public IP address 200.10.228.131 and port 80.

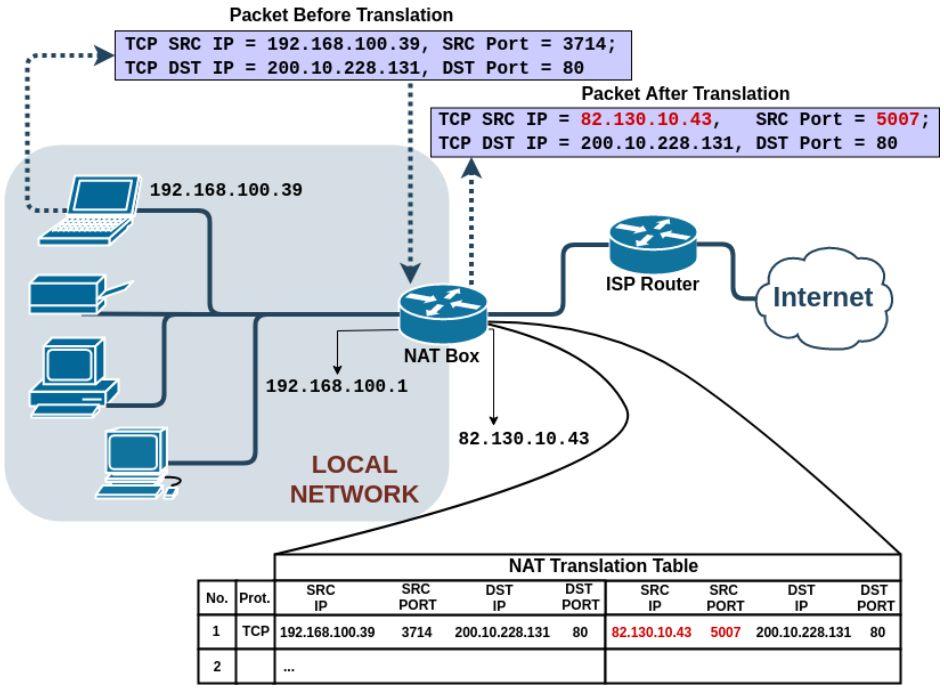

Figure 2.1: Example of a Traditional NAT behaviour.

In the diagram, we can see the outgoing packet before it leaves the premise with its source and destination IP addresses and ports. Later, the NAT box takes action, changes the internal IP source address 192.168.100.39 of the packet to the routable IP address 82.130.10.43 given by the Internet Service Provider (ISP). The NAT also changes the source port to 5007 as illustrated in the diagram.

Since the behavior of NATs lacks a unified standard, different policies may apply to the translation and assignment of ports and IP addresses [8]. We usually find the NAT box as a part of a router or firewall since the latter is responsible for controlling incoming and outgoing traffic at the boundary of the local network.

The key challenge is the inbound traffic since the local network appears as a single IP address to the external world. The NAT translation table makes it possible to keep track of the connections and map the network addresses and ports. It identifies and redirects inbound traffic to the corresponding internal

host, as shown in Figure 2.1. The NAT deletes an entry from the translation table if the data flow remains idle for a vendor-specific period of time [8].

2.1.1 Origin and motivation of NAT

The rapid expansion of the Internet in the early 1990s caused an enormous demand for IP addresses for user networks and home computers. The IP addresses allocation procedure was not only facing shortcomings but also outright depletion of IP addresses because of the high rate of growth [8].

The need for a solution was imminent, and the design of the improved version of IP, namely IPv6, was at an early stage. IPv6 was considered the long-term solution, but engineers conceived NAT as a short-term solution to meet the excessive demand [9].

The formation of the IPng working group that was later renamed to IPv6 took place in late 1994 [8]. They published the first document to standardize IPv6 as RFC 1883 in December 1995 [10], and it reached the maturity level of Internet Standard with the RFC 8200 in December 2017 [11], twenty-two years later. In contrast, the first document to outline NATs was the RFC 1631 in May 1994 [12].

2.1.2 Guidelines for NAT design

The presence of NAT brought some controversies by purists, since it breaks the initial architectural design that each host should be able to reach other hosts directly [13]. Hence, the IETF did not exert any efforts to standardize NAT, which resulted in arbitrary NAT implementations that produced undesirable and unpredictable behavior in applications [14, 15].

As NAT gained popularity, the IETF had to adopt it through publishing best practices; however, legacy NATs can still be found on the public internet. Here a brief list of those efforts:

- RFC 4787 Network Address Translation (NAT) Behavioral Requirements for Unicast UDP [15]
- RFC 5382 NAT Behavioral Requirements for TCP [16]
- RFC 5508 NAT Behavioral Requirements for ICMP [17]
- RFC 6888 Common Requirements for Carrier-Grade NATs (CGNs) [18]
- RFC 7857 Updates to Network Address Translation (NAT) Behavioral Requirements [19]

Eventually, major NAT vendors and the IETF collaboratively decided to aim for minimizing the harm of NAT on applications. The IETF provided a classification of NATs according to their behavior to assist this inconvenience.

2.1.3 Types of NATs by mapping behavior

At the first attempt, in RFC 3489, NATs were classified with terms such as "Full Cone", "Restricted Cone", "Port Restricted Cone", and "Symmetric" [20]. That terminology is currently discouraged since it brought a lot of confusion and did not describe the observed behavior of NATs in real-life.

To ease this confusion, RFC 4787 introduced the current standardized terminology based on observed NAT behaviors [15]. IETF bases the criteria of classifications on the **reuse** of the mapping of internal source IP address and port to an external IP address and port **for new sessions**.

- Endpoint-Independent Mapping (EIM):

  In this classification, the endpoint concerns any external endpoint of the NAT. The NAT will reuse the same external port mapping for packets that have the same source IP address and port. The endpoint-Independent Mapping NAT disregards the destination IP address and port when mapping but ensures this behavior for packets that have the same source IP address and port. As an example, Table 2.1 shows in blue the source IP address and source port that the NAT considers when reusing the mapping; in purple, we see the reused ports.

SRC IP	SRC PORT	<–>	SRC IP	SRC PORT
192.168.100.39	3714		82.130.10.43	5007
192.168.100.39	3714		82.130.10.43	5007
172.222.10.5	7303		82.130.10.43	2012
172.222.10.5	7303		82.130.10.43	2012

Table 2.1: Translation table for an endpoint-independent mapping NAT

- Address-Dependent Mapping:

  In address-dependent mapping, the word "address" refers to the destination IP address to which the host will communicate. The NAT will reuse the same external port mapping for packets that have the same source IP address, port, and destination IP address. The address-dependent mapping NAT disregards the destination port when mapping. As an exam-

ple, Table 2.2 shows in blue the source IP address, source port, and destination IP address that the NAT considers when reusing the mapping; in purple, we see the reused ports.

SRC IP	SRC PORT	DST IP	<-->	SRC IP	SRC PORT	DST IP
192.168.100.39	3714	200.10.228.131		82.130.10.43	5007	200.10.228.131
192.168.100.39	3714	200.10.228.131		82.130.10.43	5007	200.10.228.131
172.222.10.5	7303	106.10.248.151		82.130.10.43	3801	106.10.248.151
172.222.10.5	7303	140.82.118.3		82.130.10.43	9101	140.82.118.3

Table 2.2: Translation table for an address-dependent mapping NAT

- Address and Port-Dependent Mapping:

  In this classification, the address and port concern the destination IP address and port to which the host will communicate. The NAT will reuse the same external port mapping for packets that has the same source IP address, port, destination IP address, and port. As an example, Table 2.3 shows in blue the source IP address, source port, destination IP address, and destination port that the NAT considers when reusing the mapping. There are no reused ports in the table.

SRC IP	SRC PORT	DST IP	DST PORT	<-->	SRC IP	SRC PORT	DST IP	DST PORT
192.168.100.39	3714	200.10.228.131	80		82.130.10.43	5007	200.10.228.131	80
192.168.100.39	3714	200.10.228.131	8080		82.130.10.43	2205	200.10.228.131	8080
172.222.10.5	7303	106.10.248.151	53		82.130.10.43	3801	106.10.248.151	53

Table 2.3: Translation table for an address and port-dependent mapping NAT

2.1.4 Other classifications of NAT

Besides the classification of mapping behavior, there are other classifications like address pooling behavior, port assignment behavior, and filtering behavior [15, 14].

Address pooling behavior classifies NAT according to the capability of choosing from a pool of IP addresses to map as the external IP address on the NAT as opposed to one IP address, as shown in the previous examples.

Port assignment behavior classifies NAT according to how the ports are assigned when mapping. For example, using the same source port number from the host when mapping on the external side of the NAT is called port preservation. The difference between mapping behavior and port assignment

behavior is that the former deals with how to reuse existing mapping in NAT when creating a new session and the latter on what port to assign for the mapping. If port preservation is not possible because the port is already in use, the NAT should assign a different port in an organized fashion.

Filtering behavior classifies NAT according to which external endpoints are allowed to use the mapping placed by the NAT.

- **Endpoint-independent filtering** corresponds to endpoint-independent mapping. The NAT filters out packets that do not have a current mapping on the translation table. However, if the internal host sends packets to any external IP address, it creates a mapping in the NAT (endpoint-independent mapping). The NAT will forward any incoming packets that have as a destination that mapping.

- **Address-dependent filtering** corresponds to address-dependent mapping. The NAT will forward incoming packets to the internal host only if the host has first sent packets to that specific external IP address.

- **Address and port-dependent filtering** correspond to address and port-dependent mapping. The NAT will forward incoming packets to the internal host only if the host has first sent packets to that specific external IP address and port.

2.1.5 NAT Traversal

NAT traversal is a mechanism or technique to establish and maintain communication between hosts that lay behind NATs. The major challenge comes from the fact that hosts in the outer or public side of the NAT cannot start connections towards hosts from the inner or private side.

The IETF recommends certain behaviors for NATs to facilitate the NAT traversal. They encourage NATs to behave as *endpoint-independent mapping* along with *endpoint-independent filtering* [15]. Administrators more often configure NAT with *address and port-dependent mapping* along with *address and port-dependent filtering*. The primary reason behind this decision is a firewall-like policy to prevent inbound connections. In addition, NAT allows to hide the internal network topology of an organization from external threat actors. This restrictive behavior provides the best protection but is the most complicated NAT type to traverse [14].

In the following section, we describe the most relevant NAT traversal techniques for our case study.

2.1.6 STUN

Session Traversal Utilities for NAT (STUN) was initially introduced as a full solution for NAT traversal. In practice, it did not work for all types of NATs. On the positive side, when it worked, it did not require any changes to the NAT behavior. Now STUN is a tool used as a component in solutions that deal with NAT traversal [21].

Assuming an end-point lies behind a NAT, the major feature of STUN is informing the end-point about the IP address and port assigned to it on the public side of the NAT. In addition, since the NAT mapping expires after a specific time, STUN provides a mechanism to keep the NAT mapping alive.

The STUN protocol also attempted to provide a complete categorization of NATs for the end-point in the initial RFC 3489, but in practice it was not possible due to diverse and arbitrary NAT implementations [21, 20].

An example of STUN configuration can be the following. Two STUN agents running the STUN protocol are used. The first agent act as a client that lies behind one or more NATs. The second agent acts as a server and lies on the public internet. The client reaches the server, and the server can identify the public IP address and port from which the request is received. The server sends this information back to the client and, thus, tells the client the public IP address and port that he lies behind.

2.1.7 Hole punching

With the hole punching technique, two hosts behind NATs establish a connection with the help of a public addressable signaling server, also known as the rendezvous server. Hole punching can be done with TCP or UDP. Below, we detail UDP hole punching as it is more relevant for our case study [22].

For a representation of the operation, we name the hosts A, B, and the server S. Hosts A and B register to the server, and the server stores two endpoints for each host [23]. The saved values are the actual IP address and port from the host and the IP address and port assigned by the external side of the NAT. This registration resembles the operation from the STUN protocol.

We summarize the establishing of a connection from A to B in three steps.

- A wants to connect to B, and it requests to S the endpoints of B.
- S replies to A with the two endpoints of B, S notifies B that A is about to attempt a connection. S sends the two endpoints of A to B. At this point, A and B know each other's endpoints, and S is no longer involved.

- A sends UDP packets to both endpoints of B and establishes the connection with the endpoint that works first. Correspondingly, B does the same towards A. As a result, A and B establish a direct connection without relaying traffic to S.

The term hole punching comes from the fact that internal hosts from a NAT gateway start by sending outbound traffic and, therefore, "punch holes" in the NAT to create a new session. This hole allows the establishment of an inbound connection, as described above.

One important observation is the filtering behavior because this NAT traversal technique requires the hole punching to be synchronous. The expected filtering behavior is to drop inbound traffic that does not correspond to any mapping in NATs.

UDP hole punching is more predictable with independent mapping and filtering NATs. With the previous example, it would be enough for A to know the external port that B used to register in S. However, with the dependent mapping of addresses and port, the coordinated punching of holes is necessary.

Besides the potential problem of different filtering behavior in NATs, UDP hole punching relies on NAT to map the same internal source ports to the same external source ports. In addition, NATs cannot always guarantee port preservation; therefore, it directly affects the outcome of a UDP hole punching operation.

2.1.8 TURN

Traversal Using Relays around NAT (TURN) its a relay extension to Session Traversal Utilities for NAT (STUN). When very disruptive NATs are in between nodes to attempt a connection, NAT traversal techniques such as hole punching fail, and there is no other option than using an intermediate node to relay the communication. This relay node is typically on the public Internet to be reachable for the nodes behind NATs [24].

The most significant advantage of using TURN is that it is almost guaranteed to be successful when traversing the NAT. However, relaying the traffic on a node requires high bandwidth from the public relay node and has higher latency than a direct connection between nodes.

A typical operation of TURN comprises a TURN client requesting another node that acts as a TURN server to behave as a relay. The TURN server allocates an IP address and port for that client to use as a relay when communicating with other peers. This allocation allows the client to communicate with multiple peers. When the TURN server receives application data from

peers, before relaying data to the client, it encapsulates the data together with information on the peer who sent the data.

2.1.9 ICE

Interactive Connectivity Establishment (ICE) is a NAT traversal technique designed to avoid assumptions in the network's topology or behavior of NATs and provide a complete and reliable solution to traverse NATs [25]. ICE works with UDP but also operates with other transport protocols such as TCP. Among the many features and capabilities, we describe here the essential operation of ICE.

The endpoints that want to establish communication are named ICE agents. Each ICE agent collects a variety of candidates, which is a combination of an IP address and a port. There are many types of candidates that ICE collects; some of them directly from the physical or logical network interface, and others discovered with STUN and TURN servers.

ICE first sorts the candidates by priority, then tests them systematically to find which is the highest-priority pair of candidates that would allow the communication between the two hosts to work [14].

In the interest of our case study, we can say that ICE combines many NAT traversal techniques, including hole punching, as described previously in this chapter. When hole punching fails, ICE uses the relaying technique from TURN to find a reliable solution.

2.2 Overlay networks

An overlay network is a network built on top of an existing network [1]. As a simple example, the almost non-existent Dial-up Internet is an overlay upon the telephone network infrastructure [26].

These days, overlay networks are developed on top of the widespread TCP/IP protocol stack, specifically in the application layer. Overlay networks are popular nowadays for their capability in solving problems that require processing and distributing a vast amount of data while being scalable and affordable. Live streaming of videos is a typical example.

To function, the overlay network depends on the underlay network for essential networking operations like routing of IP packets in the case of IP. Nodes in an overlay network link logically, while in the underlay network, they might extend across many physical hops in the network.

In current communications solutions we often see the division of traffic into data plane and control plane, for instance in the session initiation protocol (SIP) or software-defined networking (SDN). Overlay networks are useful in these two scenarios. The overlay network as a data plane (also known as forwarding plane) can cooperate in forwarding and propagation of data. Overlay networks can also route control messages and achieve connectivity between nodes as control plane elements.

Even though overlay networks present a performance overhead and are not as fast as the dedicated routers which run the Internet, we can identify some benefits [1]:

1. Incremental deployment:

   The underlay network, hardware and routers do not require modifications for the overlay network. This property allows the gradual placement of nodes in the overlay network with the benefit of monitoring and controlling routing paths between nodes along the deployment.

2. Adaptability:

   The Internet infrastructure might fall behind some concerns that are application-specific. Whereas, overlay networks can be shaped to respond to this limitation based on metrics such as latency, bandwidth, and security.

3. Robustness:

   Due to the adaptable nature of overlay networks, multiple alternative paths can be provided to route around faults. When having a sufficient amount of nodes, the overlay network can overcome network and node failures. As an example, when a direct path is not available, the traffic can be forwarded to alternative ones using additional nodes. This creates a network overhead at the expense of keeping the communication between nodes.

On the other hand, overlay networks have some limitations and we can point out three central challenges [1]:

1. Reachability:

   Because of NATs and firewalls in the underlying network of TCP/IP within the Internet, there is no guarantee of end-to-end reachability. Most of the time, overlay networks are unaware of the underlying network topology and the context in which the overlay network is being

used. In consequence, overlay networks require special techniques to achieve connectivity between nodes.

2. Management and Administration:

   Overlay networks in practice require a management interface to manage the network. When more nodes and parties join the network, managing multiple domains becomes a complex task. For this reason, most overlay networks are limited to a single administrative domain. In addition, the administrator of the overlay network typically is not present in the node that performs the overlay routing, causing the need for advanced techniques in detecting and correcting fault nodes.

3. Overhead:

   Since overlay networks run commonly on top of the Internet, they cannot be as fast and efficient as the dedicated hardware that comprises the routers. The traffic of the overlay network goes through different routers and devices across the Internet. Therefore, it does not possess enough information about the underlying topology to use in favor of routing decisions costing an overhead compared to the Internet.

2.2.1 Types of secure overlays

Here we list some relevant secure overlay networks:

- **WireGuard:** WireGuard is an open-source secure VPN tunnel focused on simplicity, high performance, and ease-of-use. It aims to have higher usability than IPsec by minimizing misconfiguration-caused failures, and be more performant than user space TLS-based solutions (such as OpenVPN) by running in kernel space [27, 28].

  WireGuard ensures simplicity by being cryptographically opinionated – it lacks cipher and protocol agility by design, and uses a single cryptographic protocol from the Noise framework. The selected protocol (Noise IK) uses Curve25519 high-speed elliptic curve function, enabling lightweight and fast negotiation. It has a single round trip key exchange and inherent protection against denial of service attacks.

  Key distribution in WireGuard is inspired by OpenSSH, being agnostic about the key distribution mechanism. Any medium can be used to exchange the public keys of the peers, after which they are able to communicate.

Finally, the attack surface of WireGuard is minimized by having a simple, small codebase with less than 4000 lines of code, easy to audit and verify its security.

- **Tailscale:** Tailscale is a mesh VPN solution built on top of WireGuard. It enhances WireGuard with features such as automatic mesh generation, single sign-on (SSO), multi-factor authentication, and centralized access control [29].

  Tailscale simplifies the connection between the nodes by assigning a stable, unique IP address to each of them. The IP address stays the same regardless of the physical location of the device. Here, the devices authenticate using existing identity providers based on SAML, OAuth2, or OpenID Connect (OIDC). As a result, two-factor and multi-factor authentication mechanisms implemented by the selected identity provider can be used. In addition, to restrict access to sensitive servers, Tailscale implements a role-based access control mechanism. In particular, it has special policy files for declaring groups (called roles in role-based access control), hosts (human-readable names to refer to a particular server), and lists of rules.

  Tailscale provides closed source user-friendly applications for nearly all platforms and an open-source command-line client for Linux.

- **Tinc:** Tinc is a VPN daemon, creating secure private networks between the hosts. It enables easy configuration and scalability by automatically creating tunnels between specified endpoints [30].

  In contrast with WireGuard, Tinc operates in the user-space. Running in user-space reduces performance. On the positive side, it ensures easy portability and kernel safety against implementation errors. The most notable features of Tinc include:

  - Compression (using zlib or LZO), encryption, and authentication (using LibreSSL or OpenSSL, message authentication codes, and sequence numbers).
  - Automated full mesh routing using direct connections, without intermediate hops.
  - NAT traversal.
  - Easy expansion; adding a node implies only adding an extra configuration file.

Tinc is open-source, runs on many operating systems, and supports IPv6. In addition to hole punching to traverse the NAT, it relies traffic through nodes similar to the TURN specification.

- **ZeroTier:** ZeroTier is an open-source network virtualization platform, enabling private and secure connections between nodes. It is designed to be easy to use and run on any platform. ZeroTier aims to minimize latency by using peer-to-peer links whenever possible. It claims to have an overhead comparable with OpenVPN and IPsec, and typically consumes less than 64MB of RAM [31].

2.2.2 Mesh networking

In a mesh network or mesh topology, the nodes connect dynamically and directly. The cooperation between nodes enables many-to-many communication from a source to a desirable destination with the property that each node acts as both host and router [32].

Mesh networks do not require a particular infrastructure to operate. They adapt dynamically and self-organize with the proper configuration to function. This organization is handled by a common shared policy between all the nodes. To find the best routes from the source to the destination, a stream of data is sent across all nodes in between, and the best routes are decided based on metrics such as hops, link quality, and throughput.

Mesh networks can be used in wireless communication, wired networks [33], as well as software applications like overlay networks. In consequence, we can point a few advantages [32]:

1. Easy to install, deploy, maintain, and less expensive to operate.
2. The workload of routing and forwarding distributes dynamically across the nodes.
3. Quick reaction to failures in nodes.
4. The mesh topology is malleable and easy to change.

2.2.3 Service mesh

To describe what is a service mesh, first, we must briefly introduce Cloud-Native Applications (CNA) since it is a component that enables the service mesh capabilities. Cloud-Native Applications are systems conceived in the

cloud able to utilize the full extent of the capabilities only found in cloud computing providers, e.g., autoscaling [34].

Service mesh architecture is an application infrastructure layer on top of cloud-native applications. Since it is a relatively recent concept, we can refer to the established interpretation of William Morgan, who has the credit to originate the term service mesh:

> "A service mesh is a dedicated infrastructure layer for handling service-to-service communication. It is responsible for the reliable delivery of requests through the complex topology of services that comprise a modern, cloud-native application. In practice, the service mesh's implementation is an array of lightweight network proxies deployed alongside micro-services, without the applications needing to be aware." [35]

The service mesh is an abstraction layer that allows, among other matters, to separate security from the application, for example, using TLS to secure the communication between micro-services. Previously, securing communication with TLS was the developer's responsibility. With the given separation, developers can now focus on software development without the need to understand how the infrastructure operates and the operations teams are responsible for securing communication.

In the past decade, the combination of developers with the operations teams adopted the DevOps name. Among the various interpretations of what DevOps refers to, we provide the following definition:

> "DevOps is a collaborative and multidisciplinary effort within an organization to automate continuous delivery of new software versions, while guaranteeing their correctness and reliability." [36]

Service mesh represents an essential change in DevOps scope since previously, DevOps was mainly dealing with the management of software releases. With a service mesh, developers do not need to write code to enable capabilities that the operations teams need, and the operations teams do not need to re-compile the system to apply changes in the system.

Figure 2.2 presents the generic topology of a service mesh. Here, each service instance is augmented with a proxy instance, also known as a sidecar proxy, that handles interservice communications, security, and monitoring related issues. This communication happens at the *data plane*. On the other hand, the *control plane* manages the behavior of the proxies. The control plane is typically connected to a command-line interface, an API, or a user interface application to configure and control the app.

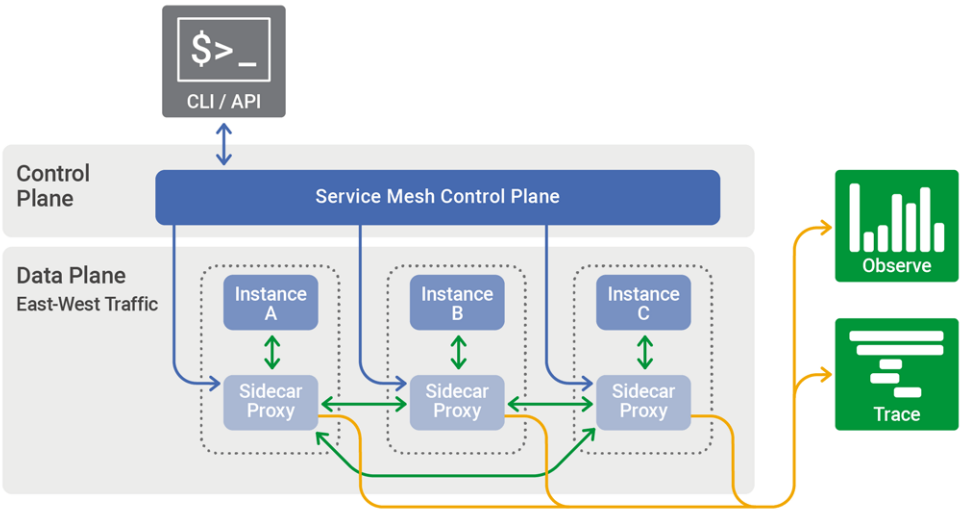

Figure 2.2: Service mesh topology [37].

2.2.4 Types of service meshes

Several open-source service mesh solutions have been developed. Below, we summarize some of them and describe the use-cases for each.

- **Linkerd:** Linkerd was the fist solution to popularize the term "service mesh". Linkerd, similar to most service meshes, provides reliability, observability, and security features. It includes in one package a control plane "Namerd" and an ultralight linkerd-proxy next to each service instance as a data plane [38].

  The solution is focused on simplicity, lightweight implementation, and low latency. In particular, Linkerd 2.x is significantly faster and smaller in size than Linkerd 1.x, as well as other, more flexible solutions such as Istio. The light weight is achieved at the price of having fewer features and supporting the Kubernetes platform only [39]. Therefore, it is best suited for minimalistic Kubernetes-based applications where no additional flexibility is required.

- **Istio:** Istio is an increasingly popular service mesh solution, originally developed by Google, IBM, and Lyft. It is a platform-agnostic solution, providing universal control plane to manage the service proxies. It pairs with a data plane, such as Envoy or Nginx proxy. Istio is designed to be portable, scalable, and support different types of applications [40].

Such flexibility and abundance of features make Istio the most widespread solution nowadays [39]. However, the flexibility comes at a price of the deployment and support complexity that may be unnecessary for applications with only basic needs.

- **Envoy:** Envoy is a high-performance application proxy for modern cloud-native applications [41]. It can act either as a standalone proxying layer or as the data plane in service mesh applications. To achieve service mesh functionality, it is typically combined with control plane providers, such as Istio as a sidecar proxy.

- **Consul:** Consul is another service mesh management framework, operating in the control plane [42]. It needs to be paired with a sidecar proxy provider such as Envoy. Consul may be the best choice for a system that needs multiple platform support, such as both Kubernetes and virtual machines, but does not need the complexity of Istio.

2.2.5 Zero trust networking

Zero trust networking is a framework based on the idea that no devices or users can be trusted regardless of being inside or outside the network perimeter [43]. It is different from traditional IT network security, where it is hard to get inside the network perimeter, but once inside, everyone is trusted and the traffic is not cryptographically protected. In zero trust networking, on the other hand, the identity of each device and user, and authorization of each action, needs to be verified.

In the cloud, zero trust networking aims to build secure network communication without relying on the physical or logical security of the underlying physical or virtual network of the cloud platform.. This approach is motivated by the fact that nowadays the information of companies is spread across the cloud servers and data centers, rather than being stored in a single location [44].

One of the fundamental principles behind zero trust networking is that the network is never trusted: there are attackers on both the inside and the outside of the network. Another principle is the requirement of authentication and authorization of both the client subject and device before establishing a session with an endpoint.

As mentioned above, zero trust networks focus on protecting the most critical resources of the network such as services, databases, and network accounts. These resources represent *protect surfaces* within the zero trust framework.

Once the protect surfaces are determined, the traffic between them is identified. This allows the cloud architects to create a micro-perimeter around each protect surface [45].

The micro-perimeter is an optimal micro-network built as an enclosure for the protect surface, aiming to provide the best-suited security. These micro-perimeters are created with a *segmentation gateway*, also called the *next generation firewall (NGFW)*. NGFW, which could be either virtual or physical, guarantees that only legitimate traffic can access the protect surface. NGFW needs to be application-aware and filter access to web services through deep packet inspection. For this reason, it focuses on Layer 7, rather than Layers 3 and 4 as in current generation firewalls [46].

Despite the presence of NGFW, attackers can attempt to disrupt the network *infrastructure* targeting Layers 3 and 4. In this case, the attacker can interfere with the routing protocol, rather than targeting a single protect surface. By this, a denial of service (DoS) can be caused. Such attacks can propagate false routing information and thus, obstruct network operations. Papadimitratos et al. [47] present an overview of approaches against routing infrastructure attacks in the Internet. In particular, they summarize methods that *prevent* such attacks and they outline methods that detect and *react* to routing protocol abuse.

2.3 IPsec

Internet Protocol Security (IPsec) is a secure network protocol suite that provides the following security services: source authentication, access control, data integrity, confidentiality via encryption, limited traffic flow confidentiality, data integrity, and replay attack prevention [48].

The IETF (Internet Engineering Task Force) knew that security was a primary concern for the growing Internet. After a long historical debate about which layer of the Internet stack should be secured, IETF settled on the idea to place the security solution in the network layer protocol [49].

IPsec is placed between the transport layer and the link layer, in practice endowing the IP protocol with security services. This decision has two primary motives:

1. Applying security on the network layer does not require modifications to existing applications.

2. End users that operate in the application layer do not need to deal with

the intricacies of cryptography and security that might lead to security breaches if not configured correctly.

2.3.1 IPsec VPN

Virtual Private Networks (VPN) are overlay networks usually running on top of public networks, i.e., the Internet, to simulate private networks [49]. VPNs provide the same properties of a typical private network, such as connectivity, QoS, and privacy by connecting two private endpoints with encryption. VPNs require a tunneling protocol such as IPsec to encapsulate the data packets from one form to another.

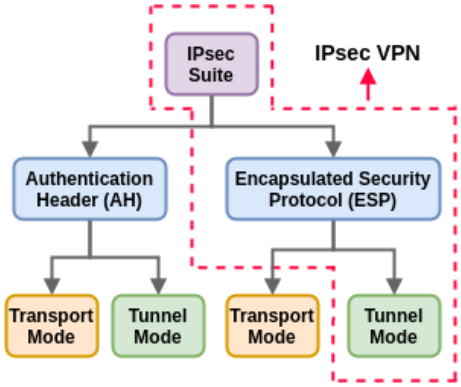

Figure 2.3: IPsec Session Protocols.

In effect, IPsec VPN is a virtual private network that uses IPsec to encrypt and encapsulate IP packets. The stream of encrypted packets forms a tunnel across the untrusted IP network [26].

There are two session protocols in the IPsec protocol suite: Authentication Header (AH) and Encapsulated Security Protocol (ESP). AH is no longer a part of the standard and IETF discourages the use of AH since ESP can provide the same security services of AH when using it with integrity and no confidentiality [48]. AH attributes its existence to the American Export controls in the 1990s as they compelled product developers to separate the support for integrity and confidentiality [50].

These two session protocols have two operation modes; the host-to-host transport mode and the network tunneling mode, as shown in Figure 2.3.

We will center the discussion of IPsec on ESP in tunnel mode. The tunneling mode is more appropriate for VPNs, and IPsec in practice is used mainly in tunnel mode.

Now we provide a description of the operation of IPsec.

IPsec VPN or IPsec with ESP in tunnel mode encapsulates and encrypts an IP packet within another IP packet. The outer IP packet possesses a standard IPv4 header that routers on the Internet can forward the datagram like any regular IP packet. The encrypted payload of the outer IP packet is another IPsec datagram that will be processed at the destination. Once the IPsec datagram arrives at the destination, the payload is decrypted and unwrapped and then passed to the upper-layer protocol, typically TCP or UDP.

2.3.2 Security associations

The communication in IPsec happens between a pair of nodes, e.g., two routers, two hosts, or a host and a router. IPsec datagrams require establishing first a security association (SA) between the pair of nodes. The SA comprises a unidirectional network-layer logical connection from the source to the destination. Typically, communication is bi-directional between the pair of nodes, which requires establishing two SAs between the nodes, one for each direction.

Both the source and the destination nodes keep the state of the SA in a Security Association Database (SAD). Each entry in the SAD represents a SA.

A typical SA includes [13]:

1. The Security Parameter Index (SPI) that acts as the identifier for the SA. SPI works as a lookup parameter in the SAD at the receiving end.

2. The origin network address of the SA and the destination network address of the SA.

3. The encryption algorithm, e.g., AES with CBC mode, NULL, AES with GCM [51].

4. The encryption and authentication keys

5. The algorithm for an integrity check, e.g., HMAC with SHA2 [51].

When the source node of communication needs to craft an IPsec datagram to the destination, it accesses the SAD to lookup which SA matches the target. After matching an SA, IPsec encrypts and authenticates the datagram with the parameters in the SAD entry. The destination node needs to keep the same settings on the SAD to apply the reverse operation of authenticating and decrypting the datagram. An IPsec node maintains many SAs in the SAD, one

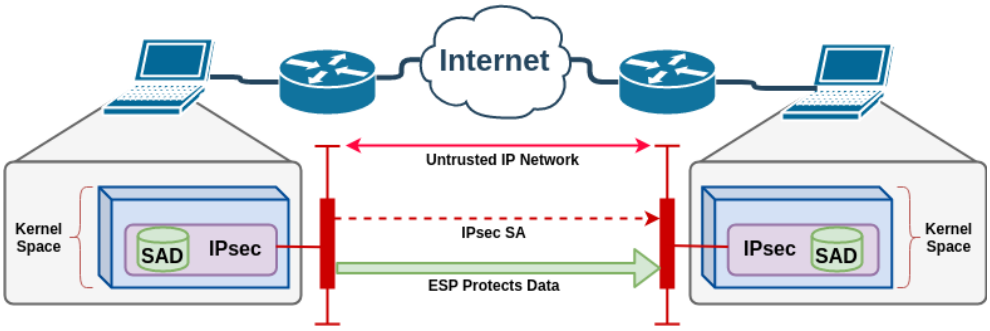

Figure 2.4: IPsec Security Association.

for each node which requires communication. The SAD is a data structure in the kernel space of the operating system.

Figure 2.4 shows the components of secure transmission of IPsec datagrams: IPsec as part of the kernel of the operating system, the correspondent SAD in the kernel space, and the unidirectional SA that enables the crafting of ESP payloads.

2.3.3 SPD policies

The source node of the secure communication can have several network interfaces, or it can be a gateway instead of an endpoint computer. In the latter case, the gateway might receive IP packets from different origins and there is no guarantee that these packets are secured with IPsec.

IPsec maintains another data structure called the Security Policy Database (SPD) to segregate IP packets that should be secure. The SPD dictates which IP packets to protect and which SA to use. To select a specific action, the SPD filters the IP packets based on the transport-layer protocol, local network address, local port, remote network address, and remote port. It can discard, bypass, or protect the matching IP packet.

2.3.4 IKEv2

We previously addressed the content of an SA and the requirement to place the same state of the SA on each of the communicating nodes. The network administrator can attain the remaining tasks, such as manually setting the SPI, keys, algorithms for authenticating, and encrypting the IP packets. However, in the actual world, IPsec VPN networks comprise hundreds or thousands of

IPsec nodes, so that manually creating SAs is impractical and difficult to manage.

The Internet Key Exchange Protocol Version 2 (IKEv2) [52], also known as IKE, is a component of IPsec used for performing mutual authentication and establishing and maintaining Security Associations (SAs). IKE provides a procedure to automate the creation of SAs and to distribute them accordingly to distant geographic locations.

The prior version of IKE, also known as IKEv1, has two phases; the first phase creates a bi-directional IKE security association (IKE SA) that is entirely different from the IPsec SA discussed in previous sections. The second phase is to establish the conventional IPsec SAs in the source and destination nodes. On the other hand, IKEv2 eliminates the usage of phases and relies on four messages (IKE_SA_INIT, IKE_AUTH, CREATE_CHILD_SA, and INFORMATIONAL). In most cases, four messages are sufficient to create the IKE SA and one child SA, however, IKE can generate several SAs if needed.

The created IKE SA arranges an authenticated and encrypted channel between the source-destination nodes. The protocol negotiates authentication and encryption algorithms, then generates the session key that the IPsec SA would use.

Certificates and public key signatures are not the only method to authenticate IKE SA. EAP or pre-shared keys between the source and destination node can also be used. When using pre-shared keys, it is crucial to assure as much randomness as the most robust key in the negotiation. Otherwise, there is room for dictionary and social-engineering attacks.

The source and destination nodes negotiate algorithms for authentication and encryption to generate the SA as described previously. The messages are signed by both nodes, therefore revealing their identity to each other through the established secure channel. Finally, the parties possess an SA in each direction and session keys for authentication and encryption of data. Using the SA and the session keys, each node can send secure messages through the IPsec tunnel.

2.3.5 Peer authorization database

The Peer Authorization Database (PAD) links the SPD and a security association management protocol, such as IKE [48]. PAD is typically needed when authenticating endpoints with certificates since IKE uses a fully qualified domain name (FQDN) as an identifier for the endpoint. SPD uses IP addresses as identifiers for selecting the policies. PAD provides a secure mapping between

the two identifier spaces, the FQDNs and the IP addresses.

PAD ensures that the resolved IP from the FQDN in the certificate corresponds in a secure to an IP address from the SPD. Overlooking this comparison can lead to some vulnerabilities [53].

2.3.6 IPsec architecture

Finally, we have all the components to illustrate the IPsec architecture, as shown in Figure 2.5. These components include three databases to manage policies and security associations, a handful of protocols like ESP, IKE, AH and various RFCs to help implementers achieve their goals. Experts have criticized IPsec for having design issues and for its deployment complexity in real-world scenarios [54]. Such complexity needs to be addressed to minimize the surface of potential vulnerabilities and to allow an attainable auditability.

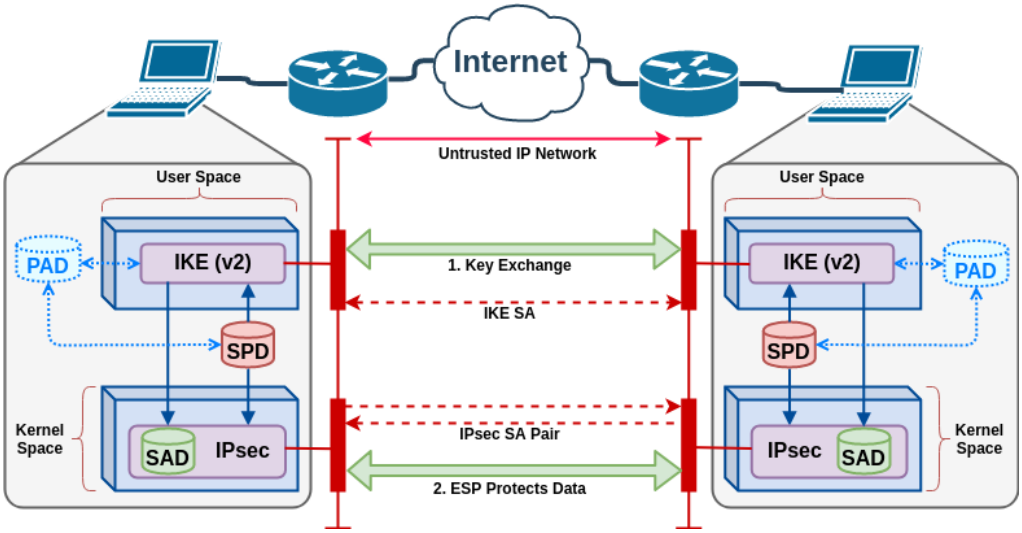

Figure 2.5: IPsec Architecture [RFC 4301] [48][55].

2.3.7 IPsec and NAT traversal

When a host attempts to communicate with IPsec AH in the presence of NATs, the NAT changes the header of packets, which causes the authentication and integrity check on the destination of the communication to fail [56]. However, in case of IPsec ESP, the NAT does not see the encrypted port numbers and cannot modify them. Therefore, IPsec encapsulates the original secure packet with a UDP header on port 4500 to traverse the NAT [57].

The newly wrapped ESP packet with a UDP header, allows the NAT to maintain port mappings and forward these packets to hosts behind the NAT [58]. The UDP encapsulation allows the packets to go through NATs, and the destination unwraps the packet to treat it accordingly by IPsec. The bottom line of NAT traversal for IPsec is UDP encapsulation. However, more operations take place to enable encapsulation, including detecting if the destination supports the NAT traversal, detecting how many NATs exists between the endpoints, and negotiating how to use UDP with IKE [59, 52].

Chapter 3

Nebula

Nebula is a scalable open-source global overlay network developed by Slack to meet the use case of server-to-server communication in multi-cloud providers, as shown in Figure 3.1. Nebula runs as an application in the user space, creates virtual interfaces, encapsulates the IP communication in UDP datagrams, and uses the hole punching technique to traverse NATs.

Before deciding to build Nebula, the developers experimented with alternatives that did not meet their expectations regarding performance, security, ease of use, and other features. Nebula took inspiration from the Tinc project for their NAT traversal approach and enhanced it with encryption, security groups, certificates, and tunneling.

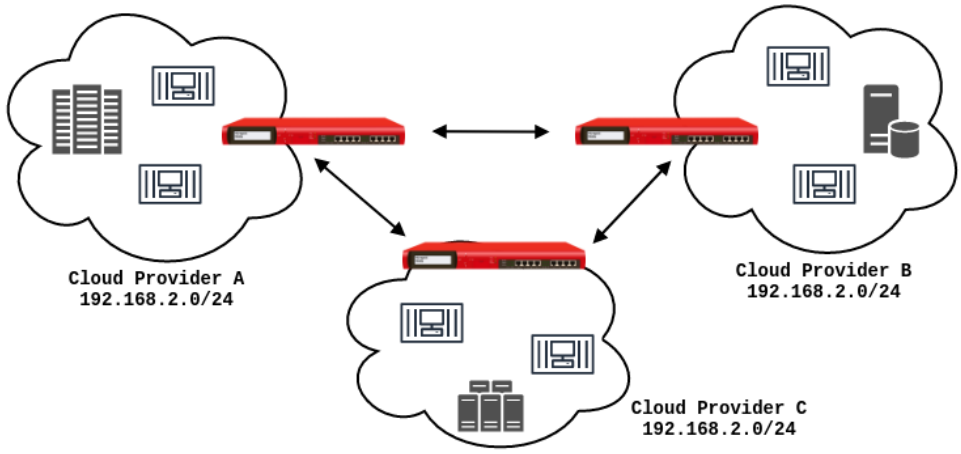

Figure 3.1: Secure communication between multi-cloud providers.

3.1 Motivation and goals

Nebula aims to achieve the following goals [60]:

- **Security:** The encryption of traffic is the most important requirement. Nebula uses the Noise cryptographic framework to secure the channels between nodes.
- **Multi-cloud and cross-platform:** As mentioned in the problem statement, Slack initially relied on IPsec to connect geographical locations in the cloud, complicating their scheme when adding new sites or regions. With an application-layer software-based solution, Nebula not only connects servers in the cloud but has the ambition to be cross-platform and connect laptops, desktops, and smartphones.
- **High-level filtering with security groups:** To avoid filtering solely by IP addresses, it is possible to use group memberships through certificates and filter the nodes by their identity. This feature is useful for ephemeral hosts in a cloud environment.
- **Strong identity:** Nebula provides authentication of nodes through non-standard certificates, along with a tool for creating a CA and signing the certificates.
- **Speed:** Initially, when using IPsec and connecting distant locations, all the traffic was relayed to an intermediate host, causing a penalty in performance. With Nebula through the hole-punch technique, if successful, the host-to-host connection is achieved directly without intermediaries.
- **Testing:** Typically, when applying filtering rules over a network, there is a risk of mistakes. Nebula allows testing filtering rules individually on nodes before extending the policy to the entire overlay network.

3.2 Architecture and overview

There are two binaries bundled with Nebula. The main application, `nebula`, and the utility to create CAs and sign certificates `nebula-cert`. The CA and the certificates have a default validity period of 365 days.

After generating the CA key pair, we create certificates for the nodes specifying the overlay network address and the membership of security groups for

that certificate. When running Nebula, we specify a configuration file as a parameter; it creates the virtual interface, and the node is ready for establishing communication.

There are two types of behaviors when running Nebula. Either the node is a lighthouse or not. When running Nebula as a lighthouse, the node acts as a rendezvous server, similar to the STUN specification. Nodes reach the lighthouse, and the lighthouse informs them about their external mapping caused by the NATs. With the help of the lighthouse, nodes can traverse NATs using the hole punching technique.

If nodes are in the same local network, they establish a connection directly. Lighthouses share nothing with other lighthouses and are used only for nodes' discovery.

The lighthouse behavior is an option among many others from the configuration file. Through configuration, one can specify the MTU, queue length, DNS server, static hosts mapping, logging, and filtering through the built-in firewall.

Nebula does not do routing per se; however, it is considered a mesh network in the sense that nodes can connect directly with the help of the lighthouse.

Figure 3.2 shows the architecture of Nebula. Light blue IP addresses represent the overlay Nebula network, whereas black color IP addresses are from the underlying network. The number 4242 corresponds to the port in which Nebula is operating. Nodes B and C are from the same network, whereas node A is in a distant location. With the overlay addresses, all the nodes can connect transparently to each other, despite the NATs.

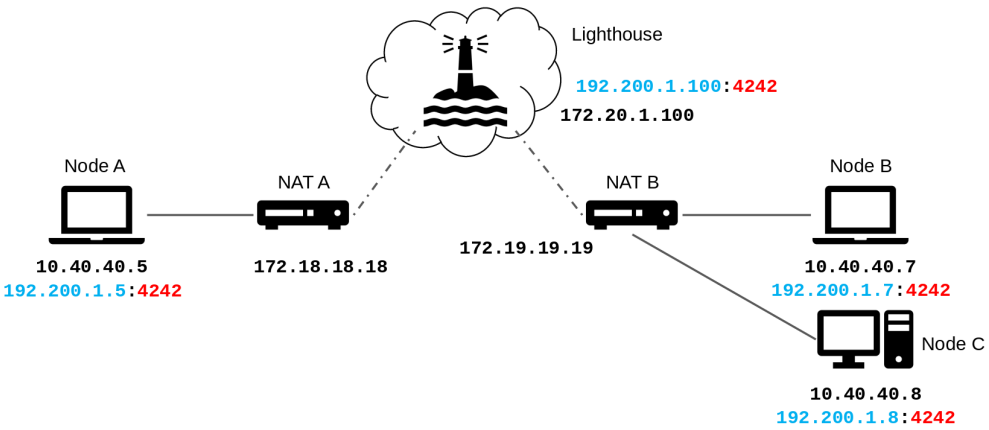

Figure 3.2: Nebula architecture.

3.3 Certificates and CA

Public key infrastructure (PKI) is a technology for managing the lifecycle of public key certificates. It uses asymmetric cryptography and can be used to provide *confidentiality* and *integrity* of data, as well as *authentication* and *non-repudiation* of entities and actions [61].

Below, we first describe several key components of the PKI. Then we specify their implementation in Nebula.

Public key certificate, or digital certificate is a digital document that ties the public key of the subject with its identity.

Certificate Authority (CA), also referred to as Certification Authority, is a trusted organization that identifies and authenticates the entities and creates a certificate signed by its private key. The CA is the foundation of the infrastructure as it is the only unit that issues certificates. The role of the CA also includes the verification of subject identities by third parties.

Certificate chain shows the path to a certificate through the chain of certificates between the root CA (also referred to as the trust anchor) and the given certificate. To verify a certificate at the end of the chain, each of the certificates in the chain needs to be verified. This is an important component of the PKI, since any of the certificates in the chain can be revoked. This hierarchy of the certificates is also known as the chain of trust.

Certificate revocation list (CRL) is used to control the status of certificates (valid, invalid, unknown). It is a digitally signed object that defines the list of the certificates that have been revoked by the CA. The list should be checked by the people or devices that are about to rely on the certificate, to ensure that it has not been revoked and can be trusted. In addition to the CRLs, an online protocol for checking the revocation status of certificates, called Online Certificate Status Protocol (OCSP), can be used. OCSP enables requesting the status of an individual certificate in real-time, rather than downloading and processing a potentially very large CRL.

Nebula uses public key certificates to verify the nodes' identity when connecting to each other. The certificate format, however, is different from the well known X.509 standard [62].

To set up a Nebula network, a certificate authority is created [63]. This CA is the root of trust for the particular network and needs to be stored in a secure place. The CA then issues certificates for each node in the network. While in a typical PKI the certificate chain can contain multiple intermediate certificates between the end-entity and the root, a Nebula certificate chain consists of only one certificate and two entities: the root CA and the node as the subject of the

certificate. The certificates include custom attributes such as security groups, environment, and roles [60].

To implement certificate revocation, Nebula uses blacklisting. Each node stores a list of certificate fingerprints that it will refuse to communicate with. Once a node is suspected to be compromised, the blacklists of all the nodes need to be updated. In addition, Nebula node certificates are recommended to have a short lifetime to reduce the attacker's time window [64].

Moreover, Nebula supports a configuration where the nodes trust multiple CAs at the same time. In this case, one of the CAs signs the certificates, while the others are stored offline for emergency situations. In case the primary CA is compromised, this will allow transitioning to signing with the new CA immediately and removing the first CA from the trusted CAs. The mechanism can also be used to substitute an expiring CA certificate [65].

3.4 Noise framework

Cryptographic protocols for secure communication might vary between applications depending on the use case. Cryptography is intrinsically complex to implement correctly, which brings the strategy to develop a general and multipurpose protocol such as the IPsec suite (e.g., gateway-to-gateway, host-to-host, etc). One alternative is having a specific and tailored secure protocol for each use case with the high risk of being undermined by security flaws or at least being prone to vulnerabilities.

The Noise framework attempts to address the gap between the general and specific protocols. Noise is not a single generic secure protocol but rather a generator of secure protocols from which implementers can choose to suit their needs when designing a networking solution. Through a simple language, combining five validity rules and a few cryptographic primitives, the Noise framework can yield protocols with specific security properties [66, 67, 68, 69].

The cryptographic primitives that the Noise framework supports are [67]:

1. A Diffie-Hellman group
2. A Hash function
3. A Key derivation function
4. An AEAD cipher

Depending on the generated protocol, the main security properties are [67]:

- Confidentiality
- Authentication and integrity
- Key compromise impersonation (KCI) resistance
- Forward-secrecy
- Resistance against replay attacks

A Noise protocol falls in the family of authenticated key exchange protocols, which are based on a Diffie-Hellman key agreement between an initiator and a responder [69]. However, there is no strict separation between the phases of key establishment and transmission of data through the channel. Depending on the generated protocol or pattern, the protocol can transmit application data even in the handshake phase as we will discuss in the following sections.

3.5 Handshake patterns

The key establishment, or also called a handshake pattern in Noise terminology, is the main contribution of the Noise framework.

The pattern language that identifies the handshake is based on Alice and Bob notation. It is worth mentioning that Alice represents the party on the left of the communication. When expressing the handshake pattern in a canonical form, it implies that Alice is the initiator while Bob is the responder. By reverting the arrows, the patterns can also be presented with Bob as the initiator without changing the handshake behavior. In the present work, for readability, we will present the patterns in the canonical form.

Interactive handshake patterns, also referred to as fundamental patterns, comprise two characters. Each character specifies how to process the static key from the initiator and responder respectively [66]. These two characters identify and define the behavior of the protocol, while other additional characters act as modifiers. This work will focus on these fundamental handshake patterns.

Here are the possible values for the first character of the pattern, i.e. initiator's static key:

- **N** = **N**o static key in use
- **K** = The static key is **K**nown to responder

- **X** = Static key for initiator **X**mitted (“transmitted”) to responder
- **I** = Static key for initiator **I**mmediately transmitted to responder, despite reduced or absent identity hiding

The second character can take the following values, i.e. responder’s static key:

- **N** = **N**o static key in use
- **K** = Static key for responder **K**nown to initiator
- **X** = Static key for responder **X**mitted (“transmitted”) to initiator

Here an example of the fundamental pattern `IN`:

```
IN:
-> e, s
<- e, ee, se
```

The handshake pattern comprises several message patterns, which contain tokens. By combining specific tokens from the following set (”e”, ”s”, “ee”, “es”, “se”, “ss”, “psk”), we deduce how the DH operations will take place. We will omit the thorough explanation of the “psk” token since it is not relevant for our study. Once all the tokens have been processed from a message pattern, Noise sends the assembled message to the other party.

Here we concisely mention the outcome of processing the tokens (see Appendix A and [66]).

- **`"e"`**: The sender generates a new ephemeral key pair (e_{pub}, e_{prv}) and places the ephemeral public key e_{pub} as a part of the message to be sent to the other party.
- **`"s"`**: The sender loads his static key pair (s_{pub}, s_{prv}) and appends his static public key s_{pub} as part of the message to be sent to the other party.
- **`"ee", "es", "se", "ss"`**: Each of these two characters describes a DH operation between the initiator and the responder. The first character corresponds to the initiator and the second character to the responder. If the character is “e“ the ephemeral key is used or a static key is used, if the character is "s".

Previously with the `IN` pattern, we identify the behavior of a noise protocol with two characters. However, we need to specify the DH function, cipher and hash function to completely qualify a noise protocol. We get the name of the protocol by spacing four underscore-separated name sections. All Noise protocols start with the ASCII string `"Noise"`.

For example, WireGuard uses the `IK` pattern in the following configurations when no pre-shared key is in use and with pre-shared key respectively [28]:

- `"Noise_IK_25519_ChaChaPoly_BLAKE2s"`
- `"NoisePSK_IK_ 25519_ChaChaPoly_BLAKE2s"`

Another example is WhatsApp that uses a **Noise Pipe** compound protocol and through the runtime analysis tool Frida, the following protocols were found [70, 71, 72]:

- `"Noise_XX_25519_AESGCM_SHA256"`
- `"Noise_IK_25519_AESGCM_SHA256"`
- `"Noise_XXfallback_25519_AESGCM_SHA256"`

Finally, Nebula uses the `IX` pattern with the option to choose the ChaCha20-Poly1305 AEAD cipher by changing the configuration file [73].

- `"Noise_IX_25519_AESGCM_SHA256"` (Default)

  ```
  IX:
  -> e, s
  <- e, ee, se, s, es
  ```

- `"Noise_IX_25519_ChaChaPoly_SHA256"`

3.6 Noise state machines

Noise uses a set of variables organized in three state machines to process the tokens, apply the rules from the handshake pattern, and keep track of the DH operations, the symmetric and session keys. Noise arranges these state machines in a hierarchy: The handshake state, the symmetric state, and the cipher state as shown in Figure 3.3.

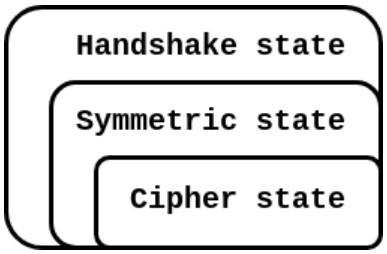

Figure 3.3: send.

3.6.1 Handshake state

Each party has a single handshake state that will be reset once they establish the communication. The variables in this state correspond to the ephemeral and static DH asymmetric keys. The handshake state keeps track of four public and two private keys [68].

- **`e`**: The name `e` stands for ephemeral, and this variable holds the ephemeral key pair internally. For readability, we denote the private and public ephemeral keys by e_{prv} and e_{pub} respectively.
- **`s`**: The name `s` stands for static, and this variable holds the static key pair internally. For readability, we denote the private and public static keys by s_{prv} and s_{pub} respectively.
- **`re`**: The name `re` stands for remote ephemeral public key, this variable holds the other party's ephemeral public key.
- **`rs`**: The name `rs` stands for remote public static key, this variable holds the other party's static public key.

3.6.2 Symmetric state

During the handshake phase, each party has a single symmetric state that will be reset once they establish the communication. The variables in this state are `h` and `ck` [66]:

- **`h`**: Variable `h` holds the hash of the protocol name. For the IX pattern used by Nebula, it is `"Noise_IX_25519_AESGCM_SHA256"`. Subsequently, with certain operations along with the handshake, the variable `h` would hash these messages concatenated with the previous values to guarantee that each party has undertaken the same steps.

- **ck**: The variable `ck` stands for the chaining key, `ck` is a hash that takes the initial value of `h`. As an input (salt) for the HKDF function [74], it derives symmetric encryption keys. The variable `ck` will contain the hashes of performed DH operations, and at the end of the handshake through the HKDF function, it will be used to derive the encryption keys for the secure transport of messages.

3.6.3 Cipher state

The variables in this state are `k` and `n`. During the handshake phase, there is a single cipher state for each participant. Later, in the transport phase, each participant has two cipher states: one to send encrypted messages and the other to decrypt receiving messages [66, 68].

- **k**: The variable `k` stands for key and holds the cipher key with which specific messages are encrypted during the handshake phase.
- **n**: The variable `n` stands for nonce but behaves like a counter. Noise uses it for encrypting and decrypting static DH keys and payloads during the handshake phase.

3.7 Functions

Below is the list of relevant functions used for processing the IX pattern. For the complete list, refer to the specification [66].

– **GEN_EPHEMEREAL_KEY_PAIR()**:

Generates the Diffie-Hellman key pair $(e_{prv},\ e_{pub})$.

Officially in the specification, the function is called `GENERATE_KEYPAIR()`, We rename it to `GEN_EPHEMEREAL_KEY_PAIR()` for better readability.

– **DH(private_key, public_key)**:

This function returns the result of a Diffie-Hellman operation between the private and public keys.

– **ENCRYPT(key, nonce, associated_data, plaintext)**:

This function returns the encryption of the fourth parameter, `plaintext`, using the given key, nonce, and associated data.

– **`DECRYPT(key, nonce, associated_data, ciphertext)`**:

  This function returns the plaintext from the ciphertext using the given key, nonce, and associated data.

– **`HKDF(chaining_key, input_key_material)`**:

  This key derivation function yields two outputs using the `chaining_key` as HKDF salt and the `input_key_material`. In Appendix A, this function is represented by a sequence of HMAC-HASH operations; for readability, we replace those operations with a single call of the HKDF function.

3.8 Overview of handshake pattern IX

In this section, we apply the processing rules of the Noise framework found in Appendix A for the pattern IX that Nebula employs. We aid the processing of messages with a visual representation of the handshake, symmetric, and cipher state machines. Finally, we conclude with a succinct representation of the messages that are sent by each party.

The tokens and messages to process are the following:

```
IX:
-> e, s
<- e, ee, se, s, es
```

First, we initialize the protocol, setting to empty the four values of the handshake state. The variable `h` from the symmetric state receives the hash value of the full protocol name, and the chaining key takes that initial value. The variable `k` from the cipher state receives an empty value and the nonce zero. Algorithm 1 shows the initialization and Figure 3.4, the visual representation of the state machines at this stage.

Note: Instead of overriding the values of the variables `h`, `k` and `ck`, we use of a subscript index with `h`, `k` and `ck` to show side to side the current values

of each party during the handshake.

Algorithm 1: Initialization of IX Pattern

Result: Set the hanshake, symmetric and cypher states to initial values

```
// initialization, these values are set in both sides
message ← "";
// handshake state
e ← ∅;
re ← ∅;
s ← ∅;
rs ← ∅;
// symmetric state
h_0 ← HASH("Noise_IX_25519_AESGCM_SHA256");
ck_0 ← h_0;
// cipher state
k ← ∅;
n ← 0;
```

Initiator

Handshake state				Symmetric state		Cypher state	
e	re	s	rs	h	ck	k	n
∅	∅	∅	∅	h_0	h_0	∅	0

Responder

Handshake state				Symmetric state		Cypher state	
e	re	s	rs	h	ck	k	n
∅	∅	∅	∅	h_0	h_0	∅	0

Figure 3.4: State machines of IX pattern during initialization

After the initialization, the pattern processes the first message from the initiator. The responder receives the ephemeral public key, the static public key, and a payload. As mentioned previously, Noise allows sending application data during the handshake phase. In the payload of this stage, Nebula sends the certificate without encryption. It is worth recalling that the IX pattern does

not encrypt the payload in the first message.

Algorithm 2: Initiator process "-> e, s"

Result: Send to the responder { $e = e_{pub},\ s = s_{pub},\ payload$ }

```
// process e
e ← GEN_EPHEMEREAL_KEY_PAIR() ; // Generates e_pub, e_prv
message ← message || e_pub;
h_1 ← HASH( h_0 || e_pub );
// process s
s ← LOAD_STATIC_KEYPAIR() ;            // Load Nebula
 certificates s_pub, s_prv
enc_s ← s_pub;
message ← message || enc_s;
h_2 ← HASH( h_1 || enc_s );
// Sends the message
ciphertext ← payload;
message ← message || ciphertext;
h_3 ← HASH( h_2 || ciphertext );
SEND(message) ;                        // e = e_pub, s = s_pub, payload
```

Algorithm 2 shows the steps of processing the message `"-> e, s"`, Algorithm Algorithm 3 shows the processing of the same message from the receiving side. Finally Figure 3.5 shows the visual representation of the state machines at this stage.

Algorithm 3: Responder process "-> e, s"

Result: Recieve from the initiator { $re = e_{pub},\ rs = s_{pub},\ payload$ }

```
// process e
( re, message ) ← split_first_component(message);
h_1 ← HASH(h_0 || re);
// process s
( enc_rs, message ) ←
 split_first_component(message);
rs ← enc_rs;
h_2 ← HASH( h_1 || enc_rs );
// process payload
payload ← message;
h_3 ← HASH( h_2 || message );
```

The responder now processes and prepares the message `"<- e, ee, se, s, es"` as shown in Algorithm 4. The responder generates the ephemeral keys and performs the DH operations with the keys received from the initiator.

Initiator

Handshake state				Symmetric state		Cypher state	
e	re	s	rs	h	ck	k	n
∅	∅	∅	∅	h_0	h_0	∅	0
e		s		h_1			
				h_2			
				h_3			

Responder

Handshake state				Symmetric state		Cypher state	
e	re	s	rs	h	ck	k	n
∅	∅	∅	∅	h_0	h_0	∅	0
	re		rs	h_1			
				h_2			
				h_3			

Figure 3.5: State machines of IX pattern after processing `"-> e, s"`

The responder sends the initiator the public ephemeral and the static keys and a payload, it then applies the HKDF function three times to get ck_3 and k_3.

Algorithm 5 shows the steps of processing the message `"<- e, ee, se, s, es"` that are received from the responder. Figure 3.6 shows the state machines holding all the last values of the IX pattern. To this point, the handshake, symmetric, and cipher states hold the same values from each party. Although there is a h_6 value in the symmetric state of the initiator, h_6 is not in

use and is just part of the algorithm generator from the Noise framework.

Algorithm 5: Initiator process "<- e, ee, se, s, es"

Result: Recieve from the responder $\{re = e_{pub},\ rs = E_{k_2}(n = 0, h_4, s_{pub}),\ payload = E_{k_3}(n = 0, h_5, \text{payload})\}$

```
// process e
( re, message ) ← split_first_component(message);
h4 ← HASH( h3 || re );
// process ee
private_key ← eprv;
public_key ← re;
dh_result ← DH(private_key, public_key);
ck1, k1 ← HKDF( ck0, dh_result );
n ← 0;
// process se
private_key ← sprv;
public_key ← re;
dh_result ← DH(private_key, public_key);
ck2, k2 ← HKDF( ck1, dh_result );
n ← 0 ;
// process s
( enc_rs, message ) ←
  split_first_component(message);
rs ← DECRYPT( k2, n, h4, enc_rs );
n ← n + 1;
h5 ← HASH( h4 || enc_rs );
// process es
private_key ← eprv;
public_key ← rs;
dh_result ← DH(private_key, public_key);
ck3, k3 ← HKDF( ck2, dh_result );
n ← 0;
// process payload
payload ← DECRYPT( k3, n, h5, message );
n ← n + 1;
h6 ← HASH( h5 || message );
```

After processing the two initial messages from the handshake, there is one last step to be performed by the participants that comprises deriving the keys to secure the channel. Algorithm 6 shows this last step by applying HKDF

Initiator

Handshake state				Symmetric state		Cypher state	
e	re	s	rs	h	ck	k	n
∅	∅	∅	∅	h_0	h_0	∅	0
e	re	s	rs	h_1	ck_1	k_1	0
				h_2	ck_2	k_2	0
				h_3	ck_3	k_3	1
				h_4			0
				h_5			1
				h_6			

Responder

Handshake state				Symmetric state		Cypher state	
e	re	s	rs	h	ck	k	n
∅	∅	∅	∅	h_0	h_0	∅	0
e	re	s	rs	h_1	ck_1	k_1	0
				h_2	ck_2	k_2	0
				h_3	ck_3	k_3	1
				h_4			0
				h_5			1

Figure 3.6: State machines of IX pattern after processing "`<- e, ee, se, s, es`"

on the last value of ck that is ck3 with an empty string as the key material. As a result, we get **k1** and **k2** keys and the nonces **n1** and **n2**. The initiator will use **k1** as a key to encrypt messages through the channel, and **k2** will be the responder's key to encrypt messages through the channel with their respective nonces. Both parties compute these two keys so they can encrypt and decrypt messages. In addition, both parties discard all other values from the state machines [68].

Algorithm 6: Key derivation

Result: Obtain the keys k1 and k2 for the secure communication

```
// Key derivation
```

1 **k1** , **k2** ← HKDF(ck_3, "");
2 **n1** ← 0;
3 **n2** ← 0;

Figure 3.7 shows a succinct representation of the Noise IX pattern used in Nebula with Alice and Bob's illustration. As components of the handshake messages, we can see the ephemeral keys and how the static keys are in transfer along with the payload. Hash equality implies key equality, and we can observe the hashes h_4 and h_5 for that purpose.

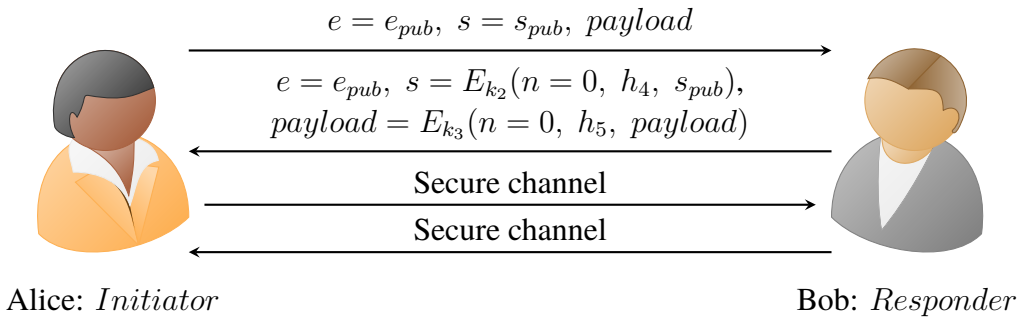

Figure 3.7: Outline of the Noise IX pattern use in Nebula

Algorithm 4: Responder process "<- e, ee, se, s, es"

Result: Send to the initiator { $e = e_{pub},\ s = E_{k_2}(n = 0, h_4, s_{pub}),\ payload = E_{k_3}(n = 0, h_5, \text{payload})$ }

// process e
$e \leftarrow$ GEN_EPHEMEREAL_KEY_PAIR() ; // Generates e_{pub}, e_{prv}
message $\leftarrow$ message $\|$ e_{pub} ;
$h_4 \leftarrow$ HASH ($h_3 \| e_{pub}$) ;
// process ee
private_key $\leftarrow$ e_{prv} ;
public_key $\leftarrow$ re ;
dh_result $\leftarrow$ DH(private_key, public_key) ;
$ck_1,\ k_1 \leftarrow$ HKDF (ck_0, dh_result) ;
n $\leftarrow 0$;
// process se
private_key $\leftarrow$ e_{prv} ;
public_key $\leftarrow$ rs ;
dh_result $\leftarrow$ DH(private_key, public_key) ;
$ck_2,\ k_2 \leftarrow$ HKDF (ck_1, dh_result) ;
n $\leftarrow 0$;
// process s
$s \leftarrow$ LOAD_STATIC_KEYPAIR() ; // Load Nebula certificates s_{pub}, s_{prv}
enc_s $\leftarrow$ ENCRYPT ($k_2,\ n,\ h_4,\ s_{pub}$) ;
n $\leftarrow$ n + 1 ;
message $\leftarrow$ message $\|$ enc_s;
$h_5 \leftarrow$ HASH ($h_4 \|$ enc_s) ;
// process es
private_key $\leftarrow$ s_{prv} ;
public_key $\leftarrow$ re ;
dh_result $\leftarrow$ DH(private_key, public_key) ;
$ck_3, k_3 \leftarrow$ HKDF (ck_2, dh_result) ;
n $\leftarrow 0$;
// Sends the message
ciphertext $\leftarrow$ ENCRYPT ($k_3,\ n,\ h_5,$ payload) ;
n $\leftarrow$ n + 1 ;
message $\leftarrow$ message $\|$ ciphertext;
SEND(message) ;
// $e = e_{pub},\ s = E_{k_2}(n = 0, h_4, s_{pub}),\ payload = E_{k_3}(n = 0, h_5, \text{payload})$

Chapter 4

Experiment

This chapter summarizes the setup of a testbed to carry experiments for the evaluation. We illustrate the network topology as a foundation for establishing secure connections with Nebula and IPsec. Then, we detail the setup of endpoints with Nebula and IPsec to finally describe the conducted experiments.

4.1 Experiment testbed setup

To test the capabilities of Nebula and IPsec, we need hosts interconnected belonging to unique network addresses. We set up a testbed with virtual machines using VirtualBox as hypervisor and Vagrant as a tool for managing and building these virtual environments. Vagrant uses boxes, which is a package format to distribute Vagrant environments in any platform in which Vagrant runs. These boxes from Vagrant are a tailored virtual machine images for a specific hypervisor that can be downloaded from the publicly available catalog online.

To provide essential tools common to all hosts for the experiment, we created two custom boxes based on "bento/ubuntu-16.04" and "bento/ubuntu-18.04" boxes for Nebula and IPsec, respectively. The network component of the virtual machines is the most crucial aspect of the experiment since we needed to have as similar behavior as possible to physical network interface cards. VirtualBox, as the hypervisor, provides several different networking operation modes for the network adapter. When creating virtual machines, Vagrant, by default, sets the network adapters to operate in the *host-only* mode. We instead specify them to run in the *internal* networking mode.

To define the membership of a network, VirtualBox would typically use IP address ranges. The Oracle VM VirtualBox driver connects the hosts by

the internal network ID to a single network switch despite them belonging to different IP address ranges, i.e., connecting by the data link layer. To overcome the problem that all virtual machines generated by Vagrant belong to the same virtual L2 network, we define our networks by designating specific internal network IDs to the adapters. As a result, only the hosts with the same internal network ID are tied by the data link layer. This allows us to reach other hosts in a different L2 network by the conventional routing in the network layer [75].

4.1.1 Architecture

Figure 4.1 shows the testbed architecture with the hosts and routers. The testbed comprises two sites: Site A and Site B, resembling the setting from the problem statement section. The sites have Gateway A and Gateway B, respectively, configured with a source NAT for hosts of the internal network to reach exterior networks. Listing 4.1 shows the iptables source NAT instruction on the external interface **eth2** from Gateways A and B.

```
# Source NAT for Gateway A
iptables --table nat --append POSTROUTING \
         --out-interface eth2 -j SNAT \
         --to-source 172.18.18.18
# Source NAT for Gateway B
iptables --table nat --append POSTROUTING \
         --out-interface eth2 -j SNAT \
         --to-source 172.19.19.19
```

Listing 4.1: Source NAT for Gateway A and B

Another VM is acting as a router between the site gateways. It simulates the functions of the Internet. On this simulated Internet, we have a host called `lighthouse1` with the network range 172.20.1.100/16, which is reachable by both sites and acts as a service in the cloud. The hosts, gateways, and the router are configured with static routes to enable the connectivity in the scheme.

Site A has the network address range 10.40.40.0/24. Site B has the same network addresses to test NAT traversal capabilities. In site A, we have a host named `node-a1` with the IP network 10.40.40.5/24, and in site B, we have a host named `node-b1` with the IP network 10.40.40.7/24. These hosts cannot interact with each other because of the NAT in place at both sites. Hosts from both sites A and B can reach `lighthouse1` without any inconvenience.

Observation: The colored rectangles in Figure 4.1 illustrate the different networks. Therefore, the words `site-a`, `site-b`, `router-network-a`, `router-network-b` and `fake-internet` correspond to the internal

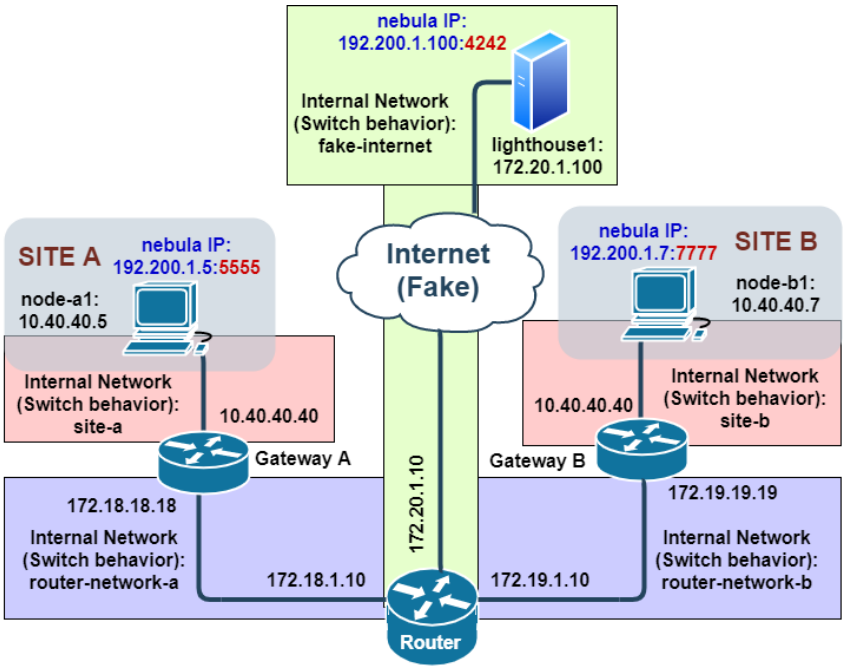

Figure 4.1: Testbed network

network ID from the Oracle VM VirtualBox driver that enables the desired switch behavior.

4.1.2 Nebula setup

In this setup, the nodes `node-a1`, `node-b1`, and `lighthouse1` are running Nebula. First, we create the certification authority (CA) with the provided binary `nebula-cert` to later generate the pair of nebula certificate and private key for the nodes, as shown in Listing 4.2.

```
./nebula-cert ca -name "Andromeda Galaxy, Inc"

./nebula-cert sign -name "lighthouse1" \
                   -ip "192.200.1.100/24"
./nebula-cert sign -name "node-a1"     \
                   -ip "192.200.1.5/24"
./nebula-cert sign -name "node-b1"     \
                   -ip "192.200.1.7/24"
```

Listing 4.2: Generating certificates for Nebula

We place these certificates in the nodes. Figure 4.1 shows the Nebula IP addresses in blue with the corresponding port in red. We list the resulting

setting below:

- **lighthouse1:**

  Real IP: 172.20.1.100, Nebula IP: 192.200.1.100, Nebula Port: 4242

- **node-a1:**

  Real IP: 10.40.40.5, Nebula IP: 192.200.1.5, Nebula Port: 5555

- **node-b1:**

  Real IP: 10.40.40.7, Nebula IP: 192.200.1.7, Nebula Port: 7777

The filtering rules for the firewall in the Nebula `config.yml` file are open to any host, protocol, and port. Listing 4.3 shows an excerpt of the configuration; for the complete configuration, see Appendix B.

```
...
outbound:
  - port: any
    proto: any
    host: any

inbound:
  - port: any
    proto: any
    host: any
```

Listing 4.3: Filtering rules for Nebula

One observation is that Nebula has complete control over the virtual network interface that creates. The default configuration file defines the maximum transfer unit (MTU) 1300 and the transmission queue length (txqueuelen) 500. To make it comparable to an IPsec setup, we set the MTU to 1500 and the txqueuelen to 1000, which are the default values assigned by a GNU/Linux OS.

4.1.3 IPsec setup

In this setup, we configured the nodes `node-a1` and `lighthouse1` with an IPsec VPN tunnel. To make it comparable to the Nebula setup, we will use X.509 certificates. Similarly, we need to create a certification authority (CA) and the certificates for the nodes. Listing 4.4 shows the commands for generating these credentials in IPsec setup.

```
# Create the CA root key
ipsec pki --gen --type rsa --size 4096 \
          --outform pem > /vagrant/config/pki/
   server-root-key.pem

# Create the self-signed CA certificate
ipsec pki --self --ca --lifetime 3650 \
 --in /vagrant/config/pki/server-root-key.pem \
 --type rsa \
 --dn "C=FI, O=VPN Server, CN=VPN Server Root CA" \
 --outform pem > /vagrant/config/pki/server-root-ca
   .pem

# We create the key for the lighthouse1
ipsec pki --gen --type rsa --size 4096 \
          --outform pem > /vagrant/config/pki/
   lighthouse1-key.pem

# We create the certificate for the lighthouse1
ipsec pki --issue --in /vagrant/config/pki/
   lighthouse1-key.pem \
 --type rsa \
 --lifetime 1825 \
 --cacert /vagrant/config/pki/server-root-ca.pem \
 --cakey /vagrant/config/pki/server-root-key.pem \
 --dn "C=FI, O=VPN lighthouse1, CN=172.20.1.100" \
 --san 172.20.1.100 \
 --flag serverAuth --flag ikeIntermediate \
 --outform pem > /vagrant/config/pki/lighthouse1-
   cert.pem

# We create the key for the node-a1
ipsec pki --gen --type rsa --size 4096
          --outform pem > /vagrant/config/pki/node-
   a1-key.pem

# We create the certificate for the node-a1
ipsec pki --issue --in /vagrant/config/pki/node-a1-
   key.pem \
 --type rsa \
 --lifetime 1825 \
 --cacert /vagrant/config/pki/server-root-ca.pem \
 --cakey /vagrant/config/pki/server-root-key.pem \
 --dn "C=FI, O=VPN node-a1, CN=10.40.40.5" \
 --san 10.40.40.5 \
 --flag serverAuth --flag ikeIntermediate \
```

```
40 --outform pem > /vagrant/config/pki/node-a1-cert.
     pem
```

Listing 4.4: Generating certificates for IPsec

To manage and configure the hosts, we use the open-source IPsec implementation strongSwan. For IKEv2 and ESP, we use the following DH group, cipher and Hash function from the IANA list of algorithms registered for IKEv2 [76, 77]:

- **DH:** Elliptic Curve 25519
- **Cipher:** 256 bit AES-GCM with 128-bit ICV
- **Hash:** HMAC-SHA-256 Truncated length 128

In this setup of IPsec VPN, `lighthouse1` acts as a VPN server and is reachable from host `node-a1` behind the NAT. For that purpose, we need to apply the following masquerade configuration in `lighthouse1` to enable communication with `node-a1`, which is a member of the network 10.40.40.0/24. Listing 4.5 shows an excerpt of the setting, for the complete configuration in `node-a1` and `lighthouse1`, see Appendix C.

```
1 ...
2 iptables -t nat -A POSTROUTING -s 10.40.40.0/24 \
3          -o eth1 \
4          -m policy --dir out --pol ipsec -j
    ACCEPT
5 iptables -t nat -A POSTROUTING -s 10.40.40.0/24 \
6          -o eth1 \
7          -j MASQUERADE
8 ...
```

Listing 4.5: Enabling IPsec connectivity towards node-a1

4.1.4 Throughput experiment

To get a measurement of the performance penalty when using IPsec and Nebula, we use the tool *iperf3*. The iperf3 tool allows active measurements of the maximum achievable bandwidth on IP networks [78].

On the `lighthouse1`, we run iperf3 as a server, and we display the measurement in megabits/sec with the following command:

```
1 iperf3 -s -f m
```

The actual measurement takes place on the client `node-a1`. We run the experiment five times for 60 seconds and calculate the throughput as an average of the measurements. When measuring the plain TCP/IP link of the testbed, we use the IP 172.20.1.100. We use the same address after placing the IPsec SA, and for Nebula, we use the IP address 192.200.1.100. We display the measurement in megabits/sec with the following commands:

```
iperf3 -t 60 -c 172.20.1.100 -f m
iperf3 -t 60 -c 192.200.1.100 -f m
```

Table 4.1 shows the measured throughput values.

Protocol	Throughput
TCP/IP	452 Mbits/sec
IPsec	360 Mbits/sec
Nebula	122 Mbits/sec

Table 4.1: Throughput measurements of plain TCP/IP, IPsec and Nebula

Further discussion takes place in Chapter 5.4.

4.1.5 Latency experiment

In this experiment, we analyze how Nebula and IPsec perform. We transfer files of different sizes with IPsec, Nebula, and plain TCP/IP, and we record the latency of the operation.

First, we generate six files with the following sizes in megabytes: 1, 10, 50, 100, 500, and 1000. We use random bytes as the content of the files with the following commands.

```
# We generate files with random content.
  head -c 1M    < /dev/urandom > /files/01
   _origin_file_0001M.dat
  head -c 10M   < /dev/urandom > /files/02
   _origin_file_0010M.dat
  head -c 50M   < /dev/urandom > /files/03
   _origin_file_0050M.dat
  head -c 100M  < /dev/urandom > /files/04
   _origin_file_0100M.dat
  head -c 500M  < /dev/urandom > /files/05
   _origin_file_0500M.dat
  head -c 1000M < /dev/urandom > /files/06
   _origin_file_1000M.dat
```

For each file size, we run the experiment 30 times, and we calculate the mean and the standard deviation. To measure the time of the operation, we use the command-line tool `time` from the GNU/Linux OS. To carry out transferring files, we use the command-line networking tools `nc (netcat)` and `scp`. We record each operation in a CSV file with the following header: `"seconds, size, filename, iteration"`.

The following lines show how the recording of the action takes place when using netcat, scp and rsync, respectively.

```
{ /usr/bin/time \
    -f "%e; ${size_of_file}; ${name_of_file};
    $iteration" \
    nc -q 0 $destination 1234 < /files/${
    name_of_files} ; } \
    2>> /results/result_${size_of_file}.csv

{ /usr/bin/time \
    -f "%e; ${size_of_file}; ${name_of_file};
    $iteration" \
    scp -q /files/${name_of_file}
    vagrant@$destination:/files/ ; } \
    2>> /results/result_${size_of_file}.csv

{ /usr/bin/time  \
    -f "%e; ${size_of_file}; ${name_of_file};
    $iteration" \
    rsync -W /files/${name_of_file} \
    rsync://$destination:12000/files/ ; } \
    2>> /results/result_${size_of_file}.csv
```

Using netcat: On the other endpoint, we need to open the port to receive incoming traffic. After each successful transfer, the netcat tool terminates the session. To keep the experiment running, we iterate 180 times the following instruction.

```
nc -l -p 1234 > /tmp/out.file
```

Using scp: To avoid prompting the password on each transfer, we first generate RSA keys and then install the public key on the destination node, i.e., `lighthouse1`.

Using rsync: We use rsync with the parameter -W to transfer the whole file and avoid the delta encoding. Also, we run rsync in daemon mode to bypass authentication and encryption.

Figure 4.2 shows the measured latency values, along with the standard deviation for each measured value; the first two plots represent the seconds of the Y-axis on a logarithm scale. Further discussion takes place in Chapter 5.4.

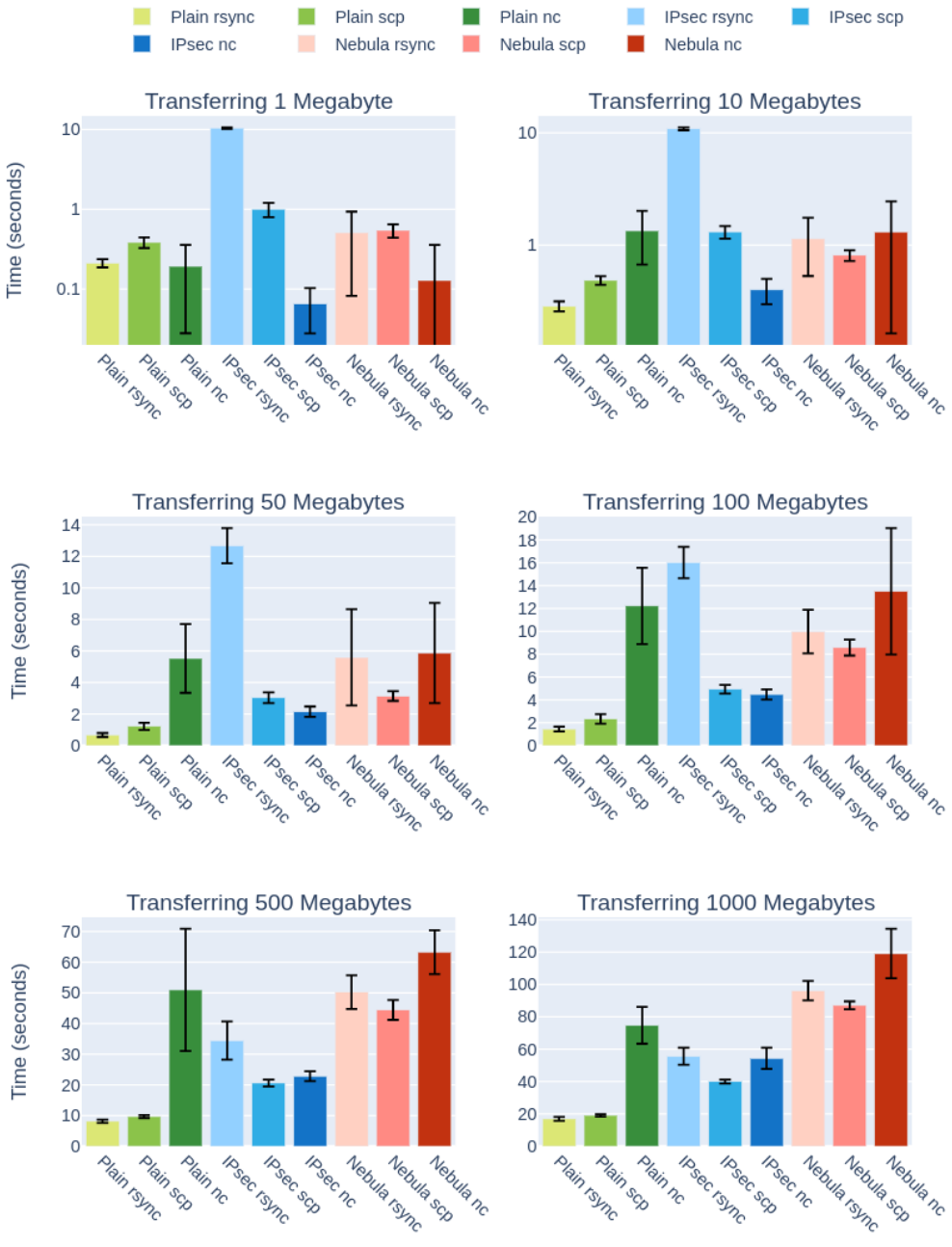

Figure 4.2: Latency experiment when transferring files.

Chapter 5

Evaluation results

This chapter analyzes the results of our evaluation of Nebula. We provide discussion and conclusions on the four aspects of assessment: reliability, security, manageability, and performance.

5.1 Reliability

In this section, we discuss the reliability of Nebula.

By design, Nebula only works in IPv4 networks. If there is a need to deploy Nebula in an IPv6 infrastructure, that would not be possible. By definition, in a mesh network, the nodes act as hosts and routers. However, Nebula does not do routing; the lighthouse is mainly for the discovery of nodes. If a link in the underlying network is down, there is no alternative route. The claim of host-to-host connectivity is one of Nebula's key features. Hence, NAT traversal must be taken into consideration in Nebula's design. In the section below, we detail our findings on how Nebula does not guarantee connectivity when nodes are behind NATs.

5.1.1 NAT traversal

The current version of Nebula (v1.2.0) applies only one (or a single) technique, called UDP hole punching. Nebula connects to hosts that are in the same network; however, connections are not guaranteed when hosts are behind NATs.

Unfortunately, we should treat NAT boxes and devices as black boxes, since their behavior might be unpredictable. This happens since some legacy NATs are still operational and do not follow current best practices.

5.1.2 Case of failure

In our experiment, we examine a failing connection attempt. As shown in our testbed design (see Figure 4.1), both gateways A and B have simple NATs in operation. We initiate the communication between **node-a1** and **node-b1**. For **node-a1** to start a secure connection, it is sufficient to send traffic towards **node-b1**. Here, from **node-a1**, we run the command: `ping -c 5 192.200.1.7` to reach **node-b1**, and the operation yields 100% of packet loss.

With no additional modification, it is not possible to achieve host-to-host connectivity in the described setup.

After this attempt, we retrieve the tracked connections from the Linux kernel with the `conntrack` tool on both gateway A and B, as shown in Table 5.1. UDP hole punching requires predictable mapping from internal to external source ports to achieve a successful connection. Table 5.1 shows that the NAT on Gateway B assigned the port 1024 instead of 7777.

Translation table in Gateway A

SRC IP	SRC PORT	DST IP	DST PORT	<-->	SRC IP	SRC PORT	DST IP	DST PORT
10.40.40.5	5555	172.20.1.100	4242		172.18.18.18	5555	172.20.1.100	4242
10.40.40.5	5555	172.19.19.19	7777		172.18.18.18	5555	172.19.19.19	7777

Translation table in Gateway B

SRC IP	SRC PORT	DST IP	DST PORT	<-->	SRC IP	SRC PORT	DST IP	DST PORT
10.40.40.7	7777	172.20.1.100	4242		172.19.19.19	7777	172.20.1.100	4242
10.40.40.7	7777	172.18.18.18	5555		172.18.18.18	**1024**	172.18.18.18	5555

Table 5.1: Translation tables on Gateway A and B

To analyze this undesired behavior, we rely on the TRACE target from iptables. On gateway B, we apply iptables policies for tracing incoming and outgoing packets to the gateway, as shown in Listing 5.1.

```
1 iptables -t raw -I PREROUTING -s 172.18.18.18 -j
    TRACE
2 iptables -t raw -I PREROUTING -s 172.20.1.100 -j
    TRACE
3 iptables -t raw -I OUTPUT -d 172.18.18.18 -j TRACE
4 iptables -t raw -I OUTPUT -d 172.20.1.100 -j TRACE
```

Listing 5.1: Tracing policies for packets

Here we describe how Nebula establishes communication between nodes. First, **node-a1** queries the lighthouse for the external IP address and port of

node-b1. The lighthouse responds to **node-a1** with the requested information but also notifies **node-b1** that **node-a1** is trying to connect. On the notification to **node-b1**, the lighthouse provides the external IP address and port of **node-a1**, so that **node-b1** can punch a hole in the NAT of Gateway B.

According to the operation of Netfilter [79] (see Appendix D) and by examining the dmesg tool logs, we observe that the incoming packet went through the following sequence of tables and chain pair: `raw:PREROUTING`, `mangle:PREROUTING`, `nat:PREROUTING`, `mangle:INPUT`, `filter:INPUT`. Finally, the gateway returns an ICMP error type 3 (Destination unreachable) code 3 (Port unreachable error) on the table and chair pair `raw:OUTPUT` [80].

As a result, we can conclude that the packet from **node-a1** arrives on gateway B before **node-b1** manages to punch a hole in the NAT. The packet from **node-a1** remains in the connection tracking of Gateway B and uses the port 7777. By the time **node-b1** punches the hole, the NAT can not preserve the port in the translation table and assigns the port 1024.

5.1.3 Workarounds

Simultaneous connection: After a few trials, we managed to achieve a successful connection without modifying the setup. This was done by starting the traffic from each node *almost* at the same time. From **node-a1** we do `ping -c 5 192.200.1.7` and from **node-b1**: `ping -c 5 192.200.1.5`. Once the connection is established, Nebula sends keep-alive packets when both nodes stop generating traffic to preserve the punched hole.

Filtering new connections: For experimental purposes, Listing 5.2 shows an iptables rule to drop new UDP connections on port 7777 for inbound packets. According to this policy, we resemble an *endpoint-independent filtering* behavior as defined in RFC 4787 [15]. Attempting the connection again from **node-a1** to **node-b1** is successful without problems.

```
1 iptables -I INPUT -p udp --dport 7777 \
2                   -m state --state NEW -j DROP
```

Listing 5.2: Filter new UDP connections on port 7777

Synchronized connection: Another approach is changing the current source code of Nebula with a synchronized attempt of connection. First, synchronizing the clocks from the nodes, then executing the normal flow while ensuring that the hole punch on the destination is accomplished *before* the packets of the originating node arrive. This is not an optimal solution since NATs might randomize ports and even use IPs from a pool of addresses. In this case, traf-

fic must be relayed as specified in the TURN and ICE standards, which is not currently implemented in Nebula.

5.1.4 Conclusion on reliability

In summary, designing a reliable mesh network with Nebula is not possible because it cannot guarantee a successful NAT traversal solution. Consequently, Nebula is only recommended in cases where full control of the network and devices can be achieved, in addition to full awareness of the network topology, similar to a data center setting.

5.2 Security

Types of credentials and credential provisioning: Nebula bases the authentication of nodes entirely on custom certificates, which are different from the X.509 standard. Nebula encodes into the certificates custom attributes, such as IPs, subnets, and groups. Custom certificates are used to ensure a compact structure that is suitable for restrictive networks with very low MTU. To create such certificates and CAs, Nebula provides the `nebula-cert` tool [81]. Both the CA created by the tool, and the certificates signed by the CA are assigned a validity period, which by default is 365 days. After the expiration, the CA cannot issue or verify certificates.

Identifier spaces and managing identities: The certificates identify Nebula nodes regardless of name or FQDN. Nebula requires the IP address and the associated (overlay) subnet in which that node would operate to be specified in the same certificate. It is also possible to blacklist any node to prevent it from interacting with other nodes, which can be done manually or using configuration management tools. The PKI section of the configuration files allows multiple CAs in place, but only one active certificate identifying the node. The possibility of having multiple CAs is useful for rotating expired CAs or refreshing the entire network with new certificates for nodes with already trusted CAs in place. This requires signing new certificates with the new CA, deploying the new certificates into the nodes with an automation tool, and removing the expired CA.

Cryptographic algorithms: One of the major advantages of Nebula when using the Noise framework is the absence of negotiation of security protocols. IKE and IPsec both require specifying the security protocol on both endpoints to establish the communication. By default, Nebula uses the elliptic

curve 25519 to perform the DH exchange, SHA-256 for hashing, and the default cipher is AES256-GCM AEAD with the option by the configuration file to change it to the ChaCha20-Poly1305 AEAD cipher. This pre-selection of security algorithms favors non-expert users, in opposition to IPsec and IKE, which require knowledge of cryptography to avoid security issues. Regarding the chosen cryptographic algorithms, the elliptic curve 25519 is one of the fastest existing curves for DH operations [82]. Both the AES-GCM cipher function and the SHA-256 hash function are recommended and standardized by NIST [83, 84].

Implementation quality: Nebula is developed in the Go programming language. After examining the source code of the project, we find out that the detailed documentation is missing for various sections of the source code. Also, external documentation does not detail most of the design choices. Only after perusing the source code, we manage to find out details regarding the default DH group, cipher suites, hash functions or the custom certificates used by Nebula.

The absence of detailed documentation complicates the auditing process for external professionals. Better documentation can ease the steep learning curve and facilitate collaboration from the community. The layout of directories within a Go project has different purposes, and their directives are enforced by the compiler. For example, the "/internal" directory forces the content not to be imported into another project. Similarly, the "pkg" directory explicitly communicates that the code within is safe to re-utilize in another project. This makes it particularly convenient to integrate the verification and signing of certificates into another tool, if the need arises. Even though the Nebula project layout structures the code in some directories, it can be improved following best practices [85]. Undoubtedly, structuring the project would require some refactoring and re-organization of the code.

5.2.1 Conclusions on security

In summary, Nebula provides certificate-based authentication, which grants a favorable balance between manageability and security for unattended server-to-server communication. However, it is the responsibility of the administrator to follow the best practices of certificate lifecycle management. IPsec provides various mechanisms for authentication, including certificates. If certificates are used for authentication then, IPsec is comparable to Nebula.

5.3 Manageability

Our main requirement is to extend a local network to a geographically distant location. Currently, an IPsec VPN connection is used to achieve this goal, which is inherently a site-to-site solution. On the other hand, Nebula is a host-to-host solution, or more precisely, a server-to-server solution. Hence, Nebula attempts to solve the problem with a different paradigm, which is creating an overlay mesh network between the hosts.

Scalability: The significant advantage of Nebula is the cloud-friendly features which IPsec does not provide.

Each node is loosely coupled in the nebula architecture since it only reports to and queries one or more lighthouse(s) to reach other nodes. New nodes can be deployed independently without affecting the entire overlay network. Moreover, nodes and lighthouses do not require any change to enable a new participant. This property makes Nebula scalable to a large number of nodes.

IPsec needs to maintain security policies between the sites explicitly, even if deployed with certificates. Adding a new site to the scheme requires applying changes to security policies of each site to enable the extension of the local network with IPsec VPN. On the other hand, secure connections in Nebula rely entirely on the session keys derived from the certificates and the handshake.

Cross cloud provider: This loosely coupled property of a Nebula node also offers mobility. Hence, one can migrate a part of the service to another cloud provider or region without compromising the security setup. The migration is done without applying any modifications to the node's certificate when Nebula is deployed as a sidecar for a microservice.

Access control: Nebula ensures access control through security groups that are implemented as a built-in firewall. The syntax for defining group memberships is similar to other cloud solutions, such as Amazon Elastic Compute Cloud (EC2) [86]. When creating certificates, it is possible to specify membership in groups as attributes to filter them with the firewall. For example, the membership of the group "webserver" can be allowed to communicate with the group "database" without the need to involve IP addresses or host names.

Besides filtering, groups can work as metadata for provisioning and configuration management tools such as chef, for example, associating a chef role with a Nebula group [87].

Logging and monitoring: Nebula has built-in support for monitoring tools, including Prometheus and Graphite. These tools are crucial for visualizing time-series data, monitoring events, alerts, and performance. While a Nebula node is operating, these monitoring tools collect and visualize data in

real-time through their interfaces.

Testing: Testing is a crucial phase of the software development cycle to guarantee a production-ready solution. Even though it is impractical to replicate a network topology for testing purposes, Nebula makes it possible to test filtering rules on singles nodes before expanding over the complete overlay network.

5.3.1 Conclusion on manageability

In summary, Nebula, is more than a conventional VPN with cloud-ready features since it is scalable and allows automation in the deployment. The most significant advantage of Nebula over IPsec is the simplicity of the administration of nodes.

5.4 Performance

In this section, we discuss the performance of Nebula concerning the research question of the project in terms of throughput and latency.

5.4.1 Throughput

Tables 4.1 and 5.2 show the performance proportions in percentages to illustrate the penalty for each communication channel. Compared with plain TCP/IP transfer, we can observe that transferring files is 20.35% slower when using IPsec. Similarly, if we compare Nebula with the other technologies, we find that it is 73.01% slower than the plain TCP/IP and 66.11% slower than IPsec.

	TCP/IP	**IPsec**	**Nebula**
TCP/IP	100.00 %	-20.35 %	-73.01 %
IPsec	-20.35 %	100.00 %	-66.11 %
Nebula	-73.01 %	-66.11 %	100.00 %

Table 5.2: Comparison of performance between plain TCP/IP, IPsec and Nebula.

This performance penalty occurs since encrypting and decrypting stream of bytes causes an additional overhead in the process of transferring files.

However, there is a significant gap in performance between IPsec and Nebula. Nebula runs entirely in the user space of the OS. In consequence, cryptographic operations take place in the user space. Doing system calls like I/O operations for accessing the network device causes context switching between user mode and supervisor mode, which is a significant reason behind the performance penalty [88].

In contrast, IPsec performs function calls directly from the supervisor mode when applying cryptographic operations and accessing the network devices, without context switching. The most significant advantage of IPsec is operating in the kernel space. As a result, IPsec is faster but less portable, whereas Nebula provides convenient portability for being a software application but with slower performance.

5.4.2 Latency

Based on the latency experiment shown in Figure 4.2, we briefly describe the tools used to transfer files. The SCP tool intrinsically creates a secure channel for transferring files. We included SCP in the experiment as a reference since SCP is useful for file transfer; however, it does not support server-to-server schemes. Besides, Netcat transfers a stream of bytes with no optimization of a specialized tool for transferring files when placed in both endpoints. Therefore, most of the time is the slowest tool on the chart. Furthermore, Rsync is a tool to synchronize files over the network; nevertheless, it can be tweaked to copy entire files to a destination. Consequently, we observe the following:

- **Observation 1:** When transferring over plain TCP/IP, Rsync is consistently faster than the other methods. Naturally, SCP has a higher latency because of the overhead caused by encryption, and it is observable from the charts. As for Netcat, it is the slowest method since it is not a specialized tool for transferring files. It does not support compression like SCP, and we suspect that it may not make maximal use of the TCP congestion window. In summary, we can rank these tools in terms of speed when transferring in plain TCP/IP as follows: Rsync, SCP, then Netcat.

- **Observation 2:** When transferring over IPsec, Rsync is drastically slower for files smaller than 100 megabytes, but is faster otherwise, as shown in the charts. Fragmentation and maximum transmission unit (MTU) size can be the technical reasons that play a role in this behavior. We can observe how Netcat achieves lower latency when sent over IPsec and UDP. Again, we hypothesize that the cause may be interaction with

TCP congestion control. Rsync over IPsec has high latency, followed by SCP for files under 500 megabytes, whereas Netcat has higher latency for files over 500 megabytes.

- **Observation 3:** When transferring over Nebula, the highest latency is achieved by Netcat, followed by Rsync, and SCP for files over 10 megabytes. This performance remains constant except for files under 1 megabyte, in which the latency ranking is inverted to the order of SCP, Rsync, and then Netcat. As shown in the chart, Nebula is expected to have the slowest transmission since it is a software application running in the user space in contrast with IPsec running in the kernel space.

- **Observation 4:** When inspecting the standard deviation for each bar in the chart, we can witness that, for SCP, it remains constant regardless of the secure channel in use. On the other hand, the variation of latency with the Netcat tool under the tunneled IPsec channel is smaller than that of the plain TCP/IP transmission, which again may depend on the tunneling of IPsec over UDP. Most of the time Nebula presents more variation than IPsec for files lower than 500 megabytes.

- **Observation 5:** When examining the different file sizes from the experiment, we can note that Nebula remains around the same threshold of 1 second when transferring files up to 10 megabytes compared to IPsec and plain TCP/IP. However, for bigger files, Nebula falls behind, and the difference is not negligible. We can observe that the slow start of the TCP congestion mechanism affects the performance of the transmission for small files independently of being tunneled with UDP. When observing the chart for transferring files of 1000 megabytes, the variations are minimal and the bars for IPsec and Nebula share the same pattern. However, as stated above, Nebula presents higher latency than IPsec.

5.4.3 Conclusion on performance

In summary, Nebula is comparable in speed to IPsec and plain TCP/IP for files under 10 megabytes. When considering server-to-server communication in which micro-services interact over HTTP or database transactions, the size of the operation typically falls under 10 megabytes. Therefore, Nebula adequately fits the use case of server-to-server communication in the cloud without a substantial penalty in performance.

Chapter 6

Conclusion

In this thesis, we evaluate Nebula, a new secure overlay mesh network. In particular, we compare its use for securing distributed applications to the traditional IPsec VPN solution. To carry out the assessment, we design a testbed with virtual machines and conduct a number of experiments. We evaluate Nebula in four key areas: reliability, security, manageability, and performance.

In terms of reliability, Nebula relies on the hole punching technique to establish connections. However, due to unpredictable NAT behaviors, the success of hole punching is not guaranteed. As Nebula lacks alternative ways of NAT traversal, it is unable to achieve host-to-host connectivity in specific NATs or NAT configurations. Thus, we discourage the use of Nebula unless the network topology is well known and controlled, for instance in a data center.

Concerning security, Nebula protects the channel with the Noise framework IX handshake pattern and applies custom certificates for secure authentication of nodes. The selected cryptographic algorithms are modern, fast, and standardized by NIST.

Concerning manageability, Nebula brings simplicity and portability. The loosely coupled design allows Nebula to scale-out seamlessly in cloud environments. One of the most significant advantages of Nebula over IPsec is the integration of the built-in firewall and the filtering by identity-based security groups.

Regarding performance, Nebula has an overhead for being an overlay network and for performing the cryptographic computations in the user-space as opposed to the kernel-space computations in IPsec. On the other hand, performing the computations in the user-space enables better portability.

In summary, Nebula brings ease of management with comparable security

but slower performance than IPsec. The thesis provides grounded results and insights for using Nebula, as well as building secure overlay networks in the future.

6.1 Future work

With the implemented testbed in this thesis, we build a foundation for extending the evaluation to other overlay networks such as Tinc, Zerotier, and Tailscale. We provide overlay network developers with an environment for testing their experimental NAT traversal solutions.

The observations in this work open two possible areas for the future work. First, the solution that Nebula provides can be enhanced and made more reliable by adding complete NAT traversal methods such as ICE. Second, the experimental setup in this work can be used to evaluate other overlay network solutions to determine if they are able to meet the requirements of this work.

Bibliography

[1] Sasu Tarkoma. *Overlay Networks: Toward Information Networking*. CRC Press, 2010. ISBN: 9781439813737. URL: https://books.google.fi/books?id=2p7MBQAAQBAJ.

[2] Christian W. Dawson. *Projects in Computing and Information Systems 2nd Edn: A Student's Guide*. 2nd. Pearson Education, 2009.

[3] Ullrich Hustadt. *Lecture: COMP516 Research Methods in Computer Science*. https://cgi.csc.liv.ac.uk/~ullrich/COMP516/notes/lect10.pdf. 2009.

[4] Bruno Marcel Duarte Coscia. *Test bed for hole punching NATs*. https://github.com/Ssruno/punching-nat-test-bed. (Accessed on 11/15/2020).

[5] Javvin Technologies Inc. *Network protocols handbook*. Saratoga, CA: Javvin Technologies, 2005. ISBN: 9780974094526.

[6] Kjeld Borch Egevang and Pyda Srisuresh. *Traditional IP Network Address Translator (Traditional NAT)*. RFC 3022. Jan. 2001. DOI: 10.17487/RFC3022. URL: https://rfc-editor.org/rfc/rfc3022.txt.

[7] Michelle Cotton et al. *Special-Purpose IP Address Registries*. RFC 6890. Apr. 2013. DOI: 10.17487/RFC6890. URL: https://rfc-editor.org/rfc/rfc6890.txt.

[8] Lixia Zhang. "A retrospective view of NAT". In: *IETF Journal* 3.2 (2007), pp. 14–20.

[9] Geoff Huston. "Anatomy: A look inside network address translators". In: *The Internet Protocol Journal* 7.3 (2004), pp. 2–32.

[10] Dr. Steve E. Deering and Bob Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 1883. Dec. 1995. DOI: 10.17487/RFC1883. URL: https://rfc-editor.org/rfc/rfc1883.txt.

[11] Dr. Steve E. Deering and Bob Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 8200. July 2017. DOI: 10.17487/RFC8200. URL: https://rfc-editor.org/rfc/rfc8200.txt.

[12] Kjeld Borch Egevang and Paul Francis. *The IP Network Address Translator (NAT)*. RFC 1631. May 1994. DOI: 10.17487/RFC1631. URL: https://rfc-editor.org/rfc/rfc1631.txt.

[13] James F Kurose and Keith W Ross. *Computer Networking: A Top-Down Approach*. Pearson Higher Education, 2017.

[14] Raimo Kantola. *S38.3115 Signaling Protocols - Lecture Notes: Network Address Translators and NAT traversal*. http://www.netlab.tkk.fi/u/kantola/Signaling/LNotes/3115-LNotes-le11.doc.pdf. 2015.

[15] Cullen Jennings and Francois Audet. *Network Address Translation (NAT) Behavioral Requirements for Unicast UDP*. RFC 4787. Jan. 2007. DOI: 10.17487/RFC4787. URL: https://rfc-editor.org/rfc/rfc4787.txt.

[16] Bryan Ford et al. *NAT Behavioral Requirements for TCP*. RFC 5382. Oct. 2008. DOI: 10.17487/RFC5382. URL: https://rfc-editor.org/rfc/rfc5382.txt.

[17] Saikat Guha et al. *NAT Behavioral Requirements for ICMP*. RFC 5508. Apr. 2009. DOI: 10.17487/RFC5508. URL: https://rfc-editor.org/rfc/rfc5508.txt.

[18] Simon Perreault et al. *Common Requirements for Carrier-Grade NATs (CGNs)*. RFC 6888. Apr. 2013. DOI: 10.17487/RFC6888. URL: https://rfc-editor.org/rfc/rfc6888.txt.

[19] Reinaldo Penno et al. *Updates to Network Address Translation (NAT) Behavioral Requirements*. RFC 7857. Apr. 2016. DOI: 10.17487/RFC7857. URL: https://rfc-editor.org/rfc/rfc7857.txt.

[20] Jonathan Rosenberg et al. *STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)*. RFC 3489. Mar. 2003. DOI: 10.17487/RFC3489. URL: https://rfc-editor.org/rfc/rfc3489.txt.

[21] Philip Matthews et al. *Session Traversal Utilities for NAT (STUN)*. RFC 5389. Oct. 2008. DOI: 10.17487/RFC5389. URL: https://rfc-editor.org/rfc/rfc5389.txt.

[22] Bryan Ford, Dan Kegel, and Pyda Srisuresh. *State of Peer-to-Peer (P2P) Communication across Network Address Translators (NATs)*. RFC 5128. Mar. 2008. DOI: 10.17487/RFC5128. URL: https://rfc-editor.org/rfc/rfc5128.txt.

[23] Bryan Ford, Pyda Srisuresh, and Dan Kegel. "Peer-to-Peer Communication Across Network Address Translators." In: *USENIX Annual Technical Conference, General Track*. 2005, pp. 179–192.

[24] Philip Matthews, Jonathan Rosenberg, and Rohan Mahy. *Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)*. RFC 5766. Apr. 2010. DOI: 10.17487/RFC5766. URL: https://rfc-editor.org/rfc/rfc5766.txt.

[25] Ari Keränen, Christer Holmberg, and Jonathan Rosenberg. *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal*. RFC 8445. July 2018. DOI: 10.17487/RFC8445. URL: https://rfc-editor.org/rfc/rfc8445.txt.

[26] J. Dong. *Network Dictionary*. ITPro collection. Javvin Press, 2007. ISBN: 9781602670006. URL: https://books.google.fi/books?id=On%5C_Hh23IXDUC.

[27] Jason A. Donenfeld. *WireGuard: fast, modern, secure VPN tunnel*. https://www.wireguard.com/. (Accessed on 28/07/2020).

[28] Jason A Donenfeld. "WireGuard: Next Generation Kernel Network Tunnel." In: *NDSS*. 2017.

[29] Tailscale Inc. *Tailscale*. https://tailscale.com/. (Accessed on 28/07/2020).

[30] Tinc VPN. *Tinc VPN*. https://www.tinc-vpn.org/. (Accessed on 28/07/2020).

[31] ZeroTier Inc. *ZeroTier – Global Area Networking*. https://www.zerotier.com/. (Accessed on 28/07/2020).

[32] Antonio Cilfone et al. "Wireless Mesh Networking: An IoT-Oriented Perspective Survey on Relevant Technologies". In: *Future Internet* 11.4 (2019), p. 99.

[33] D. Allan et al. "Shortest path bridging: Efficient control of larger ethernet networks". In: *IEEE Communications Magazine* 48.10 (2010), pp. 128–135. DOI: 10.1109/MCOM.2010.5594687.

[34] A. Khatri and V. Khatri. *Mastering Service Mesh*. Packt Publishing, 2020. ISBN: 9781789615791. URL: https://books.google.fi/books?id=B%5C_PLxgEACAAJ.

[35] William Morgan. *What's a service mesh? And why do I need one? Buoyant*. https://buoyant.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one. (Accessed on 28/05/2020). Apr. 2017.

[36] Leonardo Leite et al. "A Survey of DevOps Concepts and Challenges". In: *ACM Comput. Surv.* 52.6 (Nov. 2019). ISSN: 0360-0300. DOI: 10.1145/3359981. URL: https://doi.org/10.1145/3359981.

[37] F5 Inc. *What Is a Service Mesh? - NGINX*. https://www.nginx.com/blog/what-is-a-service-mesh. (Accessed on 28/07/2020). Apr. 2018.

[38] Linkerd Authors. *Linkerd*. https://linkerd.io/. (Accessed on 28/07/2020).

[39] Sachin Manpathak. *Kubernetes Service Mesh: A Comparison of Istio, Linkerd and Consul*. https://platform9.com/blog/kubernetes-service-mesh-a-comparison-of-istio-linkerd-and-consul/. (Accessed on 28/07/2020). Oct. 2019.

[40] Istio Authors. *Istio*. https://istio.io/. (Accessed on 28/07/2020).

[41] Envoy Project Authors. *Envoy Proxy*. https://www.envoyproxy.io/. (Accessed on 28/07/2020).

[42] HashiCorp. *Consul*. https://www.consul.io/. (Accessed on 28/07/2020).

[43] Evan Gilman and Doug Barth. *Zero Trust Networks: Building Secure Systems in Untrusted Networks*. 1st. O'Reilly Media, Inc., 2017. ISBN: 1491962194.

[44] Scott Rose et al. *Zero Trust Architecture*. Tech. rep. National Institute of Standards and Technology, 2020.

[45] S. Mehraj and M. T. Banday. "Establishing a Zero Trust Strategy in Cloud Computing Environment". In: *2020 International Conference on Computer Communication and Informatics (ICCCI)*. 2020, pp. 1–6. DOI: 10.1109/ICCCI48352.2020.9104214.

[46] K. Neupane and R. Haddad and L. Chen. "Next Generation Firewall for Network Security: A Survey". In: *SoutheastCon 2018*. 2018, pp. 1–6. DOI: 10.1109/SECON.2018.8478973.

[47] P. Papadimitratos and Z. J. Haas. “Securing the Internet routing infrastructure”. In: *IEEE Communications Magazine* 40.10 (2002), pp. 60–68. DOI: 10.1109/MCOM.2002.1039858.

[48] Karen Seo and Stephen Kent. *Security Architecture for the Internet Protocol*. RFC 4301. Dec. 2005. DOI: 10.17487/RFC4301. URL: https://rfc-editor.org/rfc/rfc4301.txt.

[49] Tanenbaum, Andrew S. and Wetherall, David J. *Computer Networks*. 5th. USA: Prentice Hall Press, 2010. ISBN: 0132126958.

[50] R.E. Smith. *Elementary Information Security*. Jones & Bartlett Learning, 2013. ISBN: 9780763761417. URL: https://books.google.fi/books?id=WrYRQi0BQDQC.

[51] Paul Wouters et al. *Cryptographic Algorithm Implementation Requirements and Usage Guidance for Encapsulating Security Payload (ESP) and Authentication Header (AH)*. RFC 8221. Oct. 2017. DOI: 10.17487/RFC8221. URL: https://rfc-editor.org/rfc/rfc8221.txt.

[52] Charlie Kaufman et al. *Internet Key Exchange Protocol Version 2 (IKEv2)*. RFC 7296. Oct. 2014. DOI: 10.17487/RFC7296. URL: https://rfc-editor.org/rfc/rfc7296.txt.

[53] Tuomas Aura, Michael Roe, and Anish Mohammed. “Experiences with Host-to-Host IPsec”. In: *Lecture Notes in Computer Science*. Vol. 4631. Springer-Verlag, Apr. 2005, pp. 3–22. DOI: 10.1007/978-3-540-77156-2_3. URL: https://www.microsoft.com/en-us/research/publication/experiences-with-host-to-host-ipsec/.

[54] Ferguson, Niels and Schneier, Bruce. *A cryptographic evaluation of IPsec*. https://www.schneier.com/wp-content/uploads/2016/02/paper-ipsec.pdf. 1999.

[55] Tuomas Aura. *CS-E4300 Network Security - Lecture: IPsec architecture*. https://mycourses.aalto.fi/course/view.php?id=24349&section=1. 2019.

[56] Dr. Bernard D. Aboba and William Dixon. *IPsec-Network Address Translation (NAT) Compatibility Requirements*. RFC 3715. Mar. 2004. DOI: 10.17487/RFC3715. URL: https://rfc-editor.org/rfc/rfc3715.txt.

[57] Tero Kivinen et al. *Negotiation of NAT-Traversal in the IKE*. RFC 3947. Jan. 2005. DOI: 10.17487/RFC3947. URL: https://rfc-editor.org/rfc/rfc3947.txt.

[58] strongSwan. *NAT Traversal (NAT-T)*. https://wiki.strongswan.org/projects/strongswan/wiki/NatTraversal. (Accessed on 11/15/2020).

[59] Victor Volpe et al. *UDP Encapsulation of IPsec ESP Packets*. RFC 3948. Jan. 2005. DOI: 10.17487/RFC3948. URL: https://rfc-editor.org/rfc/rfc3948.txt.

[60] Ryan Huber. *Introducing Nebula, the open source global overlay network from Slack*. https://slack.engineering/introducing-nebula-the-open-source-global-overlay-network-from-slack-884110a5579. (Accessed on 28/07/2020). Nov. 2019.

[61] Jeff Stapleton and W Clay Epstein. *Security without Obscurity: A Guide to PKI Operations*. CRC Press, 2016.

[62] Sharon Boeyen et al. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 5280. May 2008. DOI: 10.17487/RFC5280. URL: https://rfc-editor.org/rfc/rfc5280.txt.

[63] slackhq/nebula. *A scalable overlay networking tool with a focus on performance, simplicity and security*. https://github.com/slackhq/nebula. (Accessed on 11/15/2020).

[64] slackhq/nebula. *Blocklist best practices. Issue #304*. https://github.com/slackhq/nebula/issues/304. (Accessed on 11/15/2020).

[65] slackhq/nebula. *Documentation question: CAs. Issue #111*. https://github.com/slackhq/nebula/issues/111. (Accessed on 11/15/2020).

[66] Trevor Perrin. *The Noise Protocol Framework, July 2018. Revision 34*. https://noiseprotocol.org/noise.pdf. July 2018.

[67] Benjamin Dowling, Paul Rösler, and Jörg Schwenk. "Flexible authenticated and confidential channel establishment (fACCE): Analyzing the Noise protocol framework". In: *IACR International Conference on Public-Key Cryptography*. Vol. 12100. LNCS. Springer. 2020, pp. 341–373.

[68] Andris Suter-Dörig. “Formalizing and verifying the security protocols from the Noise framework”. Bachelor’s thesis. ETH Zürich, 2018.

[69] Guillaume Girol. “Formalizing and verifying the security protocols from the Noise framework”. MA thesis. ETH Zürich, 2019.

[70] Marvin Schirrmacher. *Analyzing WhatsApp Calls with Wireshark, radare2 and Frida by Marvin Schirrmacher. Medium*. https://medium.com/@schirrmacher/analyzing-whatsapp-calls-176a9e776213. (Accessed on 12/07/2020). Feb. 2020.

[71] WhatsApp Inc. *Whatsapp Encryption Overview - Technical white paper*. https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf. Dec. 2017.

[72] P. Rösler, C. Mainka, and J. Schwenk. “More is Less: On the End-to-End Security of Group Chats in Signal, WhatsApp, and Threema”. In: *2018 IEEE European Symposium on Security and Privacy (EuroS P)*. 2018, pp. 415–429.

[73] Yoav Nir and Adam Langley. *ChaCha20 and Poly1305 for IETF Protocols*. RFC 8439. June 2018. DOI: 10.17487/RFC8439. URL: https://rfc-editor.org/rfc/rfc8439.txt.

[74] Dr. Hugo Krawczyk and Pasi Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869. May 2010. DOI: 10.17487/RFC5869. URL: https://rfc-editor.org/rfc/rfc5869.txt.

[75] Oracle. *Oracle VM VirtualBox User Manual Version 6.1.6*. https://download.virtualbox.org/virtualbox/6.1.6/UserManual.pdf. 2020.

[76] Andreas Steffen. *IKEv2 Cipher Suites - strongSwan*. https://wiki.strongswan.org/projects/strongswan/wiki/IKEv2CipherSuites. (Accessed on 18/07/2020). Mar. 2020.

[77] Tero Kivinen. *Internet Key Exchange Version 2 (IKEv2) Parameters*. https://www.iana.org/assignments/ikev2-parameters/ikev2-parameters.xhtml. (Accessed on 18/07/2020). July 2020.

[78] ESnet and Lawrence Berkeley National Laboratory. *iPerf - The TCP, UDP and SCTP network bandwidth measurement tool*. https://iperf.fr/. (Accessed on 26/07/2020).

[79] Magnus Boye. “Netfilter connection tracking and NAT implementation”. In: *Seminar on Network Protocols in Operating Systems*. Aalto University; Aalto-yliopisto, 2013.

[80] J. Postel. *Internet Control Message Protocol*. RFC 792. Sept. 1981. DOI: 10.17487/RFC0792. URL: https://rfc-editor.org/rfc/rfc792.txt.

[81] Nathan Brown. *Feature Request: Support for RFC 8410 Ed25519 certificates and keys · Issue #51 · slackhq/nebula*. https://github.com/slackhq/nebula/issues/51. (Accessed on 28/07/2020). Dec. 2019.

[82] Daniel J Bernstein. “Curve25519: new Diffie-Hellman speed records”. In: *International Workshop on Public Key Cryptography*. Vol. 3958. LNCS. Springer. 2006, pp. 207–228.

[83] NIST. *Secure Hash Standard (SHS). Federal Information Processing Standards Publication 180-1*. 1995.

[84] Morris J. Dworkin. *SP 800-38D. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. Special Publication. Gaithersburg, MD, USA: National Institute of Standards & Technology, 2007.

[85] Golang Standards Project. *golang-standards/project-layout: Standard Go Project Layout*. https://github.com/golang-standards/project-layout. (Accessed on 07/28/2020).

[86] Amazon Elastic Compute Cloud. *Amazon EC2 security groups for Linux instances*. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-security-groups.html. (Accessed on 11/15/2020).

[87] Guest: Ryan Huber. *Podcast LINUX Unplugged 329 by Jupiter Broadcasting: Flat Network Truthers*. https://linuxunplugged.com/329?t=1936. Nov. 2019.

[88] Tanenbaum, Andrew S and Bos, Herbert. *Modern operating systems*. Pearson, 2015.

[89] Jan Engelhardt. *Packet flow in Netfilter and General Networking*. https://commons.wikimedia.org/wiki/File:Netfilter-packet-flow.svg. (Accessed on 25/07/2020). May 2019.

Appendix A

Processing tokens in Noise framework

```
message_buffer ← ''
FOR token IN message.tokens
        IF token = 'e'
                e ← GENERATE_KEYPAIR()
                message_buffer ← message_buffer || e.public_key
                h ← HASH(h || e.public_key)
                IF pattern_contains_psk
                        temp ← HMAC-HASH(ck, e.public_key)
                        ck ← HMAC-HASH(temp, 0x01)
                        k ← HMAC-HASH(temp, ck || 0x02)
                        n ← 0
        ELSEIF token = 's'
                s ← load_static_keypair()
                IF k = empty
                        enc_s ← s.public_key
                ELSE
                        enc_s ← ENCRYPT(k, n, h, s.public_key)
                        n ← n + 1
                message_buffer ← message_buffer || enc_s
                h ← HASH(h || enc_s)
        ELSEIF token = 'psk'
                psk ← load_corresponding_psk()
                temp ← HMAC-HASH(ck, psk)
                ck ← HMAC-HASH(temp, 0x01)
                temp_h ← HMAC-HASH(temp, ck || 0x02)
                k ← HMAC-HASH(temp, temp_h || 0x03)
                n ← 0
                h ← HASH(h || temp_h)
        ELSE
                IF token = 'ee'
                        private_key ← e.private_key
                        public_key ← re
                ELSEIF token = 'ss'
                        private_key ← s.private_key
                        public_key ← rs
                ELSEIF (token = 'es') = i_am_alice
                        private_key ← e.private_key
                        public_key ← rs
                ELSE
                        private_key ← s.private_key
                        public_key ← re
                dh_result ← DH(private_key, public_key)
                temp ← HMAC-HASH(ck, dh_result)
                ck ← HMAC-HASH(temp, 0x01)
                k ← HMAC-HASH(temp, ck || 0x02)
                n ← 0
IF k = empty
        ciphertext ← payload
ELSE
        ciphertext ← ENCRYPT(k, n, h, payload)
        n ← n + 1
message_buffer ← message_buffer || ciphertext
h ← HASH(h || ciphertext)
send(message_buffer)
```

Figure A.1: Initiator processing of tokens in the Noise framework [68]

```
message_buffer ← receive()
FOR token IN message.tokens
        IF token = 'e'
                (re, message_buffer) ←
                        split_first_component(message_buffer)
                h ← HASH(h || re)
                IF pattern_contains_psk
                        temp ← HMAC-HASH(ck, re)
                        ck ← HMAC-HASH(temp, 0x01)
                        k ← HMAC-HASH(temp, ck || 0x02)
                        n ← 0
        ELSEIF token = 's'
                (enc_rs, message_buffer) ←
                        split_first_component(message_buffer)
                IF k = empty
                        rs ← enc_rs
                ELSE
                        rs ← DECRYPT(k, n, h, enc_rs)
                        n ← n + 1
                h ← HASH(h || enc_rs)
        ELSEIF token = 'psk'
                psk ← load_corresponding_psk()
                temp ← HMAC-HASH(ck, psk)
                ck ← HMAC-HASH(temp, 0x01)
                temp_h ← HMAC-HASH(temp, ck || 0x02)
                k ← HMAC-HASH(temp, temp_h || 0x03)
                n ← 0
                h ← HASH(h || temp_h)
        ELSE
                IF token = 'ee'
                        private_key ← e.private_key
                        public_key ← re
                ELSEIF token = 'ss'
                        private_key ← s.private_key
                        public_key ← rs
                ELSEIF (token = 'es') = i_am_alice
                        private_key ← e.private_key
                        public_key ← rs
                ELSE
                        private_key ← s.private_key
                        public_key ← re
                dh_result ← DH(private_key, public_key)
                temp ← HMAC-HASH(ck, dh_result)
                ck ← HMAC-HASH(temp, 0x01)
                k ← HMAC-HASH(temp, ck || 0x02)
                n ← 0
IF k = empty
        payload ← message_buffer
ELSE
        payload ← DECRYPT(k, n, h, message_buffer)
        n ← n + 1
h ← HASH(h || message_buffer)
```

Figure A.2: Responder processing of tokens in the Noise framework [68]

Appendix B

Nebula configuration files

```
pki:
  ca: /vagrant/files/ca.crt
  cert: /vagrant/files/lighthouse1.crt
  key: /vagrant/files/lighthouse1.key
static_host_map:
  "192.200.1.100": ["172.20.1.100:4242"]
lighthouse:
  am_lighthouse: true
  interval: 60
  hosts:
  remote_allow_list:
    192.168.0.0/16: false
  local_allow_list:
    interfaces:
      eth0: false
      10.40.40.0/24: false
listen:
  host: 172.20.1.100
  port: 4242
punchy:
  punch: true
  respond: true
  delay: 1s
tun:
  dev: nebula1
  drop_local_broadcast: false
  drop_multicast: false
  tx_queue: 1000
  mtu: 1500
  routes:
  unsafe_routes:
```

```
logging:
  level: debug
  format: text
firewall:
  conntrack:
    tcp_timeout: 120h
    udp_timeout: 3m
    default_timeout: 10m
    max_connections: 100000
  outbound:
    - port: any
      proto: any
      host: any
  inbound:
    - port: any
      proto: any
      host: any
```

Listing B.1: config.yml file fore 'lighthouse1'

```
pki:
  ca: /vagrant/files/ca.crt
  cert: /vagrant/files/node-a1.crt
  key: /vagrant/files/node-a1.key
static_host_map:
  "192.200.1.100": ["172.20.1.100:4242"]
lighthouse:
  am_lighthouse: false
  interval: 60
  hosts:
    - "192.200.1.100"
  remote_allow_list:
    192.168.0.0/16: false
  local_allow_list:
    interfaces:
      eth0: false
      10.40.40.0/24: false
listen:
  host: 10.40.40.5
  port: 5555
punchy:
  punch: true
  respond: true
  delay: 1s
tun:
  dev: nebula1
  drop_local_broadcast: false
  drop_multicast: false
```

```
  tx_queue: 1000
  mtu: 1500
  routes:
  unsafe_routes:
logging:
  level: debug
  format: text
firewall:
  conntrack:
    tcp_timeout: 120h
    udp_timeout: 3m
    default_timeout: 10m
    max_connections: 100000
  outbound:
    - port: any
      proto: any
      host: any
  inbound:
    - port: any
      proto: any
      host: any
```

Listing B.2: config.yml file fore 'node-a1'

```
pki:
  ca: /vagrant/files/ca.crt
  cert: /vagrant/files/node-b1.crt
  key: /vagrant/files/node-b1.key
static_host_map:
  "192.200.1.100": ["172.20.1.100:4242"]
lighthouse:
  am_lighthouse: false
  interval: 60
  hosts:
    - "192.200.1.100"
  remote_allow_list:
    192.168.0.0/16: false
  local_allow_list:
    interfaces:
      eth0: false
      10.40.40.0/24: false
listen:
  host: 10.40.40.7
  port: 7777
punchy:
  punch: true
  respond: true
  delay: 1s
```

```
tun:
  dev: nebula1
  drop_local_broadcast: false
  drop_multicast: false
  tx_queue: 1000
  mtu: 1500
  routes:
  unsafe_routes:
logging:
  level: debug
  format: text
firewall:
  conntrack:
    tcp_timeout: 120h
    udp_timeout: 3m
    default_timeout: 10m
    max_connections: 100000
  outbound:
    - port: any
      proto: any
      host: any
  inbound:
    - port: any
      proto: any
      host: any
```

Listing B.3: config.yml file fore 'node-b1'

Appendix C

IPsec configuration files

```
# Copy certificates
cp /vagrant/config/pki/server-root-ca.pem   /etc/ipsec.d/cacerts/
cp /vagrant/config/pki/lighthouse1-key.pem  /etc/ipsec.d/private/
cp /vagrant/config/pki/lighthouse1-cert.pem /etc/ipsec.d/certs/
cp /vagrant/config/pki/node-a1-cert.pem     /etc/ipsec.d/certs/

# sudo ufw allow 500,4500/udp

sudo iptables -t nat -A POSTROUTING -s 10.40.40.0/24 -o eth1 -m policy --dir out --pol ipsec -j ACCEPT
sudo iptables -t nat -A POSTROUTING -s 10.40.40.0/24 -o eth1 -j MASQUERADE

cat >/etc/ipsec.conf <<EOL
config setup
    charondebug="all"
    strictcrlpolicy=no
    uniqueids=yes
conn lighthouse1-to-node-a1
    type=tunnel
    forceencaps=yes
    auto=start
    dpdaction=clear
    keyexchange=ikev2
    ike=aes256gcm128-sha256-curve25519!
    esp=aes256gcm128-sha256-curve25519!
```

```
    left=172.20.1.100
    leftid="C=FI, O=VPN lighthouse1, CN=172.20.1.100"
    leftcert=/etc/ipsec.d/certs/lighthouse1-cert.pem
    leftsubnet=0.0.0.0/0
    right=%any
    rightid="C=FI, O=VPN node-a1, CN=10.40.40.5"
    rightsourceip=10.40.40.5
    rightcert=/etc/ipsec.d/certs/node-a1-cert.pem
EOL


cat >/etc/ipsec.secrets <<EOL
172.20.1.100 : RSA "/etc/ipsec.d/private/lighthouse1-key.pem"
EOL
```

Listing C.1: IPsec configuration in lighthouse1

```
# Copy certificates
cp /vagrant/config/pki/server-root-ca.pem   /etc/ipsec.d/cacerts/
cp /vagrant/config/pki/node-a1-key.pem      /etc/ipsec.d/private/
cp /vagrant/config/pki/node-a1-cert.pem     /etc/ipsec.d/certs/
cp /vagrant/config/pki/lighthouse1-cert.pem /etc/ipsec.d/certs/


cat >/etc/ipsec.conf <<EOL
config setup
    charondebug="all"
    strictcrlpolicy=no
    uniqueids=yes
conn node-a1-to-lighthouse1
    type=tunnel
    forceencaps=yes
    auto=start
    dpdaction=clear
    keyexchange=ikev2
    ike=aes256gcm128-sha256-curve25519!
    esp=aes256gcm128-sha256-curve25519!
    right=172.20.1.100
    rightid="C=FI, O=VPN lighthouse1, CN=172.20.1.100"
    rightsubnet=172.20.1.0/24
```

```
24      rightcert=/etc/ipsec.d/certs/lighthouse1-cert.
    pem
25      leftsourceip=%config
26      leftid="C=FI, O=VPN node-a1, CN=10.40.40.5"
27      leftcert=/etc/ipsec.d/certs/node-a1-cert.pem
28  EOL
29
30  cat >/etc/ipsec.secrets <<EOL
31  10.40.40.5 : RSA "/etc/ipsec.d/private/node-a1-key.
    pem"
32  EOL
```

Listing C.2: IPsec configuration in node-a1

Appendix D

Packet flow in Netfilter

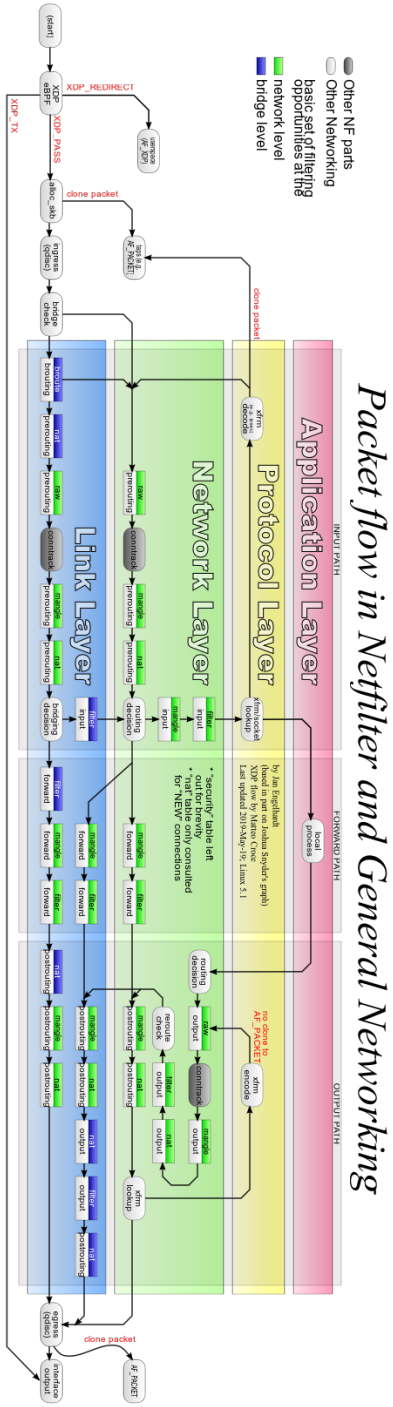

Figure D.1: Netfilter packet flow [89]

TRITA-EECS-EX-2020:912