Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2023

Vasco Regal Sousa

Multiple Client WireGuard Based Private and Secure Overlay Network

DOCUMENTO PROVISÓRIO

"An idiot admires complexity,
a genius admires simplicity."

— Terry A. Davis

o júri / the jury

presidente / president

ABC
Professor Catedrático da Universidade de Aveiro (por delegação da Reitora da Universidade de Aveiro)

vogais / examiners committee

DEF
Professor Catedrático da Universidade de Aveiro (orientador)

GHI
Professor associado da Universidade J (co-orientador)

KLM
Professor Catedrático da Universidade N

agradecimentos / acknowledgements

Ágradecimento especial aos meus gatos

Desejo também pedir desculpa a todos que tiveram de suportar o meu desinteresse pelas tarefas mundanas do dia-a-dia

Abstract

An overlay network is a group of computational nodes that communicate with each other through a logical channel, built on top of another network. Nodes in an overlay network, which are generally end systems, run Internet applications capable of performing more complex operations besides just forwarding and switching traffic. As such, an overlay network can apply routing rules and data manipulation to create a custom protocol running on top of another network. This document presents a secure and private overlay network solution deployed over University of Aveiro's very restrictive network. This solution allows clients, namely the autonomous robots residing in the Intelligent Robotics and Systems Laboratory research unit, to establish direct communications with each other, regardless of their physical location within the campus or network security mechanisms which render such connections impossible. This not only aids developers as they can interact directly with the robots through their personal machines, but also makes it possible to deploy solutions capable of communicating through any of the campus' buildings.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Network security has become a topic of growing interest in any information system. Companies strive to ensure their communications follow principles of integrity and confidentiality while minimizing attack vectors that could compromise services and data. With such goals in mind, network topologies are subjected to traffic constraints and security mechanisms to protect their systems.

Such is the case in the University of Aveiro (UA), where the network, although covering most of the campus' area, enforces several constraining mechanisms that prevent, for example, the establishment of direct Peer to Peer (P2P) connections between two clients.

The Intelligent Robotics and Systems Laboratory (IRIS-Lab) is a research unit operating in UA's premises which develops projects using autonomous mobile robots, capable of communicating through a wireless network. Currently, the robots are confined to the laboratory's internal network, since, as mentioned above, UA's highly restrictive network prevents the robots from communicating directly using UA's network, also referred to as **eduroam**.

Overcoming these limitations would be extremely valuable for IRIS-Lab's developments. In fact, allowing robots to communicate directly in a P2P communication would not only enable solutions across multiple buildings but also aid researchers during development, as they would be able to interact with the robots directly through their personal machines.

1.2 Objectives

This dissertation aims to implement a private overlay network manager to be used exclusively by UA's clients. The concept of an overlay network entails the creation of a communication layer built on top of an already existing network.

In the IRIS-Lab scenario, the management platform should provide operations to achieve a secure, private communication between a group of robots connected to UA's network, regardless of their physical location within the campus. Moreover, the authentication and connection to a desired overlay network by the robots must be a seamless operation, requiring little to no manual configuration.

To reinforce the privacy and confidentiality, this solution should be hosted entirely within UA's premises, preferably using open source tools.

Therefore, the objectives for this dissertation can be summarized as (i) enabling secure P2P communications between clients connected to UA's network, (ii) automation of client deployment, authentication and configuration mechanisms, creating an abstraction layer for the usual robot operations and (iii) ensure communication overhead is suitable for IRIS-Lab's projects requirements.

1.3 Document Structure

This document presents an implementation of such an overlay network manager. Hence, it is structured in two main chapters, the state of the art and the methodology. The former describes an exploration of the background and current state of the art, providing an analysis not only of potential tools, protocols, and frameworks suitable for the scope of the dissertation but also of published research conducted covering similar topics and scenarios. The latter establishes the work methodology to be taken for the development and results gathering process.

Chapter 2

Background and State of the Art

"O caos é uma ordem por decifrar"

— José Saramago

2.1 Overlay Networks

In the last few decades, the Internet has been subjected to an exponential growth, both in the number of users and connected devices. To answer the increasing demand and support aspects such as mobility and scalability, Internet applications have diverged from classic distributed systems to more complex network topologies, creating an extremely heterogeneous environment. In such a non-patterned landscape, P2P overlay networks have emerged as a topic of growing interest, as conducted research on the matter attempts to create networking solutions capable of addressing the adversities imposed by the modern day Internet. This section explores the fundamental principles of overlay networks and how its abstraction layer is able to produce a topology with the potential to bring a logical order to the chaotic network architecture of the Internet.

By definition, an overlay network is a logical network implemented on top of the links of another, already established, underlay network [1]. In other words, nodes (also called peers) in an overlay network (which also exist in the underlay network) implement its own application-defined routing and datagram processing behaviour. Hence, the Internet application running in the nodes is responsible for the creation and management of the P2P logical links that form the overlay network. Figure 2.1 illustrates this concept.

Overlay networks can be used in applications to achieve numerous functionalities, such as finding alternate routes for unicast applications, explored by Resilient Overlay Networks (RON) [2], which measures peers' average latencies in order to discover routes optimizing network end-to-end reliability and performance. This idea is also the principle behind Software-Defined WAN (SD-WAN) services, where forwarding rules are dynamically reconfigured and automated according to a set of defined policies, an emerging paradigm with prominent results [3, 4]. Another common example of the use of overlay networks are object sharing applications such as BitTorrent, where the overlay network is formed by the nodes which possess parts of the desired file. When a peer wants to download a file, it establishes P2P connections to the other overlay peers and retrieves the part of the data until the file is complete [5].

While decentralized by design, some topologies applying overlay concepts have a degree of

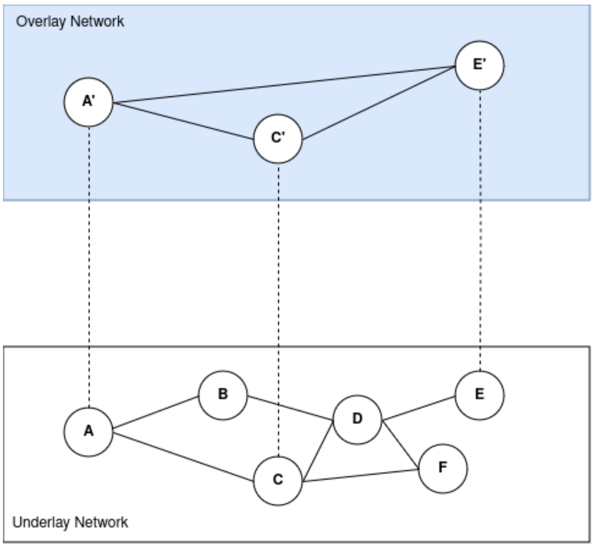

Figure 2.1: Concept of a very basic overlay network. Nodes A, C and E create logical links with each other, forming an overlay network

centralization, generally serving information regarding active peers in the network. Although a central point can cause bottlenecks, on-demand peer information is able to make a system easier to scale and reconfigure. BitTorrent is an example of this, where a new peer queries a central server, called a tracker, retrieving the addresses and identification of the peers it should connect to. Such centralization can, however, be discarded by implementing a peer finding algorithm which relies solely on the other peers already in the network.

As the nodes in an overlay network are systems running Internet applications, they are generally capable of performing more computational demanding operations than simply forwarding traffic, which is the case when dealing with devices in the underlay network, like routers or switches. Routing traffic through a node allows any application-defined manipulation to be applied to the datagrams, namely cryptographic operations. This idea is explored in further sections, as it serves as the backbone in several encrypted P2P communication protocols.

Summarizing, the use of overlay networks provides an efficient and reliable way to create a logical structure between a set of nodes residing in an otherwise unstructured topology. Additionally, as overlay nodes are essentially end systems, there's great leverage for applications to freely manipulate and route traffic. By relying on P2P communications, overlay networks' decentralized nature provides a very scalable and reconfigurable design.

2.2 Virtual Private Networks

The Internet is built on public infrastructure users generally have no control over. Public routers, relay nodes, servers and even physical links always carry the risk of traffic eavesdropping and tampering by untrusted entities. Therefore, to ensure the confidentiality, privacy and integrity of connections established through an insecure medium, data should be properly encrypted and authenticated. Such is the fundamental principle of Virtual Private Networks (VPNs).

Conceptually, a VPN is a virtual network that provides functionalities which secure the transmission of data between any two endpoints [6]. This section aims to analyze how VPN architectures have evolved, from its inception to more recent paradigms, which follow overlay concepts discussed in Section 2.1. Then, it presents an overview of notable VPN providers and their respective pros, cons and suitability for the scope of this solution. This state of the art survey aims to pickup on similar works [7, 8], while additionally reviewing emerging protocols.

2.2.1 VPN architectures

Traditional VPN services operate under a hub-and-spoke architecture, a model composed by one or more VPN Gateways - devices accepting incoming connections from client nodes and forwarding the traffic to their final destination. Hub-and-spoke architectures carry some well known bottlenecks [9, 10] and, more recently, [11]. First, it implies increased latency associated with the geographical distance between a client and the nearest hub. This topology also raises reliability issues as hubs introduce a single point of failure to the system, requiring backup plans and alternative routes to ensure no network down times.

To avoid the congestion associated with hub-and-spoke architectures, decentralized topologies have emerged which operate by establishing direct P2P connection between clients, creating a mesh network.

VPNs can be classified according to their topology in two main categories: client-to-site and site-to-site. A client-to-site VPN is characterized by connections from a single user (client) to a private network (site), while site-to-site VPNs offer a secured connection between two private networks. Thus, in site-to-site networks, users are not required to individually configure VPN clients. The tunnel in this type of VPN is made available to the entire network.

2.2.2 VPNs on NAT Networks

A Network Address Translator (NAT) is a networking mechanism responsible for translating Internet Protocol (IP) addresses in private networks to public addresses when packets sent from a private network are routed to the public Internet. In the context of VPN communications, this process can prove to be a major constraint, not only due to NAT's tampering of IP packets' fields, namely destination and source addresses, which could potentially compromise its integrity in the eyes of a VPN protocol, but also regarding the dynamically changing public IP addresses which NAT decides to translate private addresses to. In other words, without additional tools, a host in a private network has no knowledge regarding which public IP it will be assigned.

In fact, it is very likely that devices on the internet reside in a network behind both NAT mechanisms and Firewall rules, with no open ports. Also, believing nodes will have a consistent static IP is a very naive assumption, especially when considering mobile devices. NAT Traversal is a networking technique that enables the establishing and maintaining (by keeping NAT holes open) of P2P connections between two peers, no matter what's standing between them, making communication possible without the need for firewall configurations or public-facing open ports. There's no one solution to achieve this functionality. In fact, there are various developments effectively implementing a NAT Traversal solution, such as ICE [12] and STUN [13]. Hence, each VPN service can have its own way of supporting NAT Traversal. Each case is explored separately.

2.3 VPN Providers

2.3.1 IPSec

IPSec refers to an aggregation of layer 3 protocols that work together to create a security extension to the IP protocol by adding packet encryption and authentication. Conceptually, IPSec presents two main dimensions: the protocol defining the transmitted packets' format, when security mechanisms are applied to them, and the protocol defining how parties in a communication negotiate encryption parameters.

Communication in an IPSec connection is managed according to Security Associations (SAs). A SA is an unidirectional set of rules and parameters specifying the necessary information for secure communication to take place [14]. Here, unidirectional means a SA can only be associated with either inbound or outbound traffic, but never with both. Hence, an IPSec bidirectional association implies the establishment of two SAs: one for incoming packets and one for outgoing. SAs specify which security mechanism to use - either Authentication Header (AH) or Encapsulating Security Payload (ESP) - and are identified by a numeric value, the Security Parameter Index (SPI). Although SAs can be manually installed in routers, gateways or machines, it becomes impractical as more clients appear. Internet Key Exchange (IKE) [15] is a negotiation protocol that tackles the problems associated with manual SA installation. In fact, IKE allows the negotiation of SA pairs between any two machines through the use of asymmetric keys or shared secrets.

Transport and Tunnel modes

IPSec supports two distinct modes of functionality: transport and tunnel [14], which differ in the way traffic is dealt with and processed. In the context of VPNs, tunnel mode presents the most desirable characteristics. First, tunnel mode encapsulates the original IP packet, allowing the use of private IP addresses as source or destination. Tunnel mode creates the concept of an "outer" and "inner“ IP header. The former contains the addresses of the IPSec peers, while the latter contains the real source and destination addresses. Moreover, this very same encapsulation adds confidentiality to the original addresses.

Transport mode requires fewer computational resources and, consequently, carries less protocol overhead. It does not, however, provide much security compared to tunnel mode, so, in the context of VPNs, tunnel mode's total protection and confidentiality of the encapsulated IP packet carry much more valuable functionalities.

Authentication Header

AH is a protocol in the IPSec suite providing data origin validation and data integrity consisting in the generation of a checksum via a digest algorithm [16]. Additionally, besides the actual message under integrity check, two other parameters are used under the AH mechanism. First, to ensure the message was sent from a valid origin, AH includes a secret shared key. Then, to ensure replay protection, it also includes a sequence number. This last feature is achieved with the sender incrementing a sequence integer whenever an outgoing message is processed.

AH, as the name suggests, operates by attaching a header to the IP packets, containing the message's SPI, its sequence number, and the Integrity Check Value (ICV) value. This last field is then verified by receivers, which calculate the packet's ICV on their end. The packet is only considered valid if there's a match between the sender and receiver's ICV.

Where this header is inserted depends on the mode in which IPSec is running. In transport mode, the AH appears after the IP header and before any next layer protocol or other IPSec headers. As for tunnel mode, the AH is injected right after the outer IP header.

To calculate the ICV, the AH requires the value of the source and destination addresses, which raises an incompatibility when faced with networks operating with NAT mechanisms [17].

Encapsulating Security Payload

The ESP protocol also offers authentication, integrity and replay protection mechanisms. It differs from AH by also providing encryption functionalities, where peers in a communication use a shared key for cryptographic operations. Analogous to the previous protocol, the ESP's header location differs in different IPSec modes. In transport mode, the header is inserted right after the IP header of the original packet. Also, in this mode, since the original IP header is not encrypted, endpoint addresses are visible and might be exposed. As for tunnel mode, a new IP header is created, followed by the ESP header.

Tunnel mode ESP is the most commonly used IPSec mode. This setup not only offers original IP address encryption, concealing source and destination addresses, but also supports the addition of padding to packets, hampering cipher analysis techniques. Moreover, it can be made compatible with NAT and employ NAT-traversal techniques [18, 19].

2.3.2 OpenVPN

OpenVPN [20] is yet another open-source VPN provider, known for its portability among the most common operating systems due to its user-space implementation. OpenVPN uses established technologies, such as Secure Sockets Layer (SSL) and asymmetric keys for negotiation and authentication and IPSec's ESP protocol, explored in the previous section, over UDP or TCP for data encryption.

TUN and TAP interfaces

OpenVPN's virtual interfaces, which process outgoing and incoming packets, have two distinct types: TUN (short for internet TUNnel) and TAP (short for internet TAP). Both devices work quite similarly, as both simulate P2P communications. They differ on the level of operation, as TAP operates at the Ethernet level. In short, TUN allows the instantiation of IP tunnels, while TAP instantiates Ethernet tunnels.

OpenVPN flow

When a client sends a packet through a TUN interface, it gets redirected to a local OpenVPN server. Here, the server performs an ESP transformation and routes the IP packet to the destination address, through the "real" network interfaces.

Similarly, when receiving a packet, the OpenVPN server will perform decipherment and validation operations on it, and, if the IP packet proves to be valid, it is sent to the TUN interface.

This process is analogous when dealing with TAP devices, differing, as mentioned before, at the protocol level.

2.3.3 WireGuard

WireGuard [21] is an open-source UDP-only layer 3 network tunnel implemented as a kernel virtual network interface. WireGuard offers both a robust cryptographic suite and transparent session management, based on the fundamental principle of secure tunnels: peers in a WireGuard communication are registered as an association between a public key (analogous to the OpenSSH keys mechanism) and a tunnel source IP address.

One of WireGuard's selling points is its simplicity. In fact, compared to similar protocols, which generally support a wide range of cryptographic suites, WireGuard settles for a singular one. Although one may consider the lack of cipher agility as a disadvantage, this approach minimizes protocol complexity, increasing security robustness by avoiding vulnerabilities commonly originating from such protocol negotiation [22].

Routing

Peers in a WireGuard communication maintain a data structure containing their own identification (both the public and private keys) and interface listening port. Then, for each known peer, an entry is present containing an association between a public key and a set of allowed source IPs.

This structure is queried both for outgoing and incoming packets. To encrypt packets to be sent, the structure is consulted, and, based on the destination address, the desired peer's public key is retrieved. As for receiving data, after decryption (with the peer's own keys), the structure is used to verify the validity of the packet's source address, which, in other words, means checking if there's a match between the source address and the allowed addresses present on the routing structure.

Optionally, WireGuard peers can configure one additional field, an internet endpoint, defining the listening address where packets should be sent. If not defined, the incoming packet's source address is used.

Cipher Suite

As mentioned before, WireGuard offers a single cipher suite for encryption and authentication mechanisms in its ecosystem. The peers' pre-shared keys are Curve25519 points, an implementation of an elliptic-curve-Diffie-Hellman function, characterized by its strong conjectured security level - presenting the same security standards as other algorithms in public key cryptography - while achieving record computational speeds [23]. Additionally, WireGuard's Curve25519 computation uses a verified implementation [24], which contributes further to the robustness of the protocol's cryptography.

Regarding payload data cryptography, a WireGuard message's plain text is encrypted with the sender's public key and a nonce counter, using ChaCha20Poly1305, a Salsa20 variation [25]. The ChaCha cryptographic family offers robust resistance to crypto-analytic methods [26], without sacrificing its state-of-the-art performance.

Finally, before any encrypted message exchange actually happens, WireGuard enforces a 1-Round Trip Time (RTT) handshake for symmetric key exchange (one for sending, and one for receiving). The messages involved in this handshake process follow a variation of the Noise

	Peer A	**Peer B**
Private Key	gIb/+...+uF2Y=	aFov...G3l0=
Public Key	FeQI...jHgE=	sg0X...7kVA=
Internet Endpoint	192.168.100.4	192.168.100.5
WireGuard Port	51820	51820

Table 2.1: Nodes to be configured with a WireGuard tunnel

[27] protocol, which is essentially a state machine controlled by a set of variables maintained by each party in the process.

Security

On top of its robust cryptographic specification, WireGuard includes in its design a set of mechanisms to further enhance protocol security and integrity.

In fact, WireGuard presents itself as a silent protocol. In other words, a WireGuard peer is essentially invisible when communication is attempted by an illegitimate party. Packets coming from an unknown source are just dropped, with no leak of information to the sender.

Additionally, a cookie system is implemented in an attempt to mitigate Distributed Denial Of Service (DDOS) attacks. Since, to determine the authenticity of a handshake message, a Curve25519 multiplication must be computed, a CPU intensive operation, a CPU-exhaustion attack vector could be exploited. Cookies are introduced as a response message to handshake initiation. These cookie messages are used as a peer response when under high CPU load, which is then in turn attached to the sender's following messages, allowing the requested handshake to proceed when the overloaded peer has available resources to continue.

Basic WireGuard Configuration

Connecting two peers in a WireGuard communication can be done with minimal configuration. After the generation of an asymmetric key pair and the setup of a WireGuard interface, it is only required to add the other peer to the routing table with its public key, allowed IPs and, optionally, its internet endpoint (where it can be currently found). After both peers configure each other, the tunnel is established and packets can be transmitted through the WireGuard interface. In a practical scenario, given two peers, *A* and *B*, with pre-generated keys and internet interfaces, presented on table 2.1, the CLI steps to setup a minimal WireGuard communication, as specified in the official WireGuard documentation are presented in figure 2.2.

WireGuard Topologies

Having outlined how a basic WireGuard scenario can be configured, the following paragraphs aim to explore four primary WireGuard topologies, which serve as the foundations for most of WireGuard's use cases.

Point to Point

```
# Peer A - interface setup
$ ip link add wg0 type WireGuard
$ ip addr add 10.0.0.1/24 dev wg0
$ wg set wg0 private-key ./private
$ ip link set wg0 up

# Adding peer B to known peers
$ wg set wg0 peer sg0X...7kVA=
    allowed-ips 10.0.0.2/32
    endpoint 192.168.100.5:51820
$
```

```
# Peer B - interface setup
$ ip link add wg0 type WireGuard
$ ip addr add 10.0.0.2/24 dev wg0
$ wg set wg0 private-key ./private
$ ip link set wg0 up

# Adding peer A to known peers
$ wg set wg0 peer FeQI...jHgE=
    allowed-ips 10.0.0.1/32
    endpoint 192.168.100.4:51820
$
```

Figure 2.2: Basic WireGuard Communication Between Two Peers

Limitations

Although WireGuard proves to be a robust, performant and maintainable protocol for encrypted communication, it still presents some complexity regarding administration agility and scalability, since it (intentionally) lacks a coordination entity. New clients added to a stand-alone WireGuard network imply the manual reconfiguration of every other peer already present, a process with added complexity and prone to errors, if carried out manually.

There are, however, several software products implementing such an orchestrator, which provide functionalities to easily manage WireGuard peers and respective tunnels, such as Tailscale, Netbird or Netmaker. These solutions are explored in further sections.

2.3.4 Protocol Comparison

The concept of performance in VPN applications refers to the impact the protocol has on network behaviour. This dimension can be empirically measured, by analysing metrics like latency, also referred to as RTT, throughput and jitter. CPU utilization is also a valuable metric, as demanding operations can also take a toll on overall network performance, since processing traffic might stall due to the CPU overloading.

The performance claims on [21], where, when comparing WireGuard to other discussed technologies like OpenVPN and IPSec, present results in favour of WireGuard in such metrics. This conclusion is backed by more extensive research [28], [29], where communication is tested in a wide range of different environments and CPU architectures.

WireGuard, due to its kernel implementation (compared to, for example, OpenVPN's user space implementation) and efficient multi-threading usage, contribute greatly to such performance benchmarks. Overall, WireGuard combines simplicity with state of the art cryptography and efficiency.

2.3.5 Summary

This section presents the fundamentals of VPNs and how they are able to ensure integrity and confidentiality to communication through insecure mediums. Then, some of the most notable VPN providers were reviewed and compared, regarding functionalities, behaviour when dealing with network constraints and overall performance. The result of this analysis suggests WireGuard as the most promising VPN protocol, due to its ease of management and configuration, security mechanisms and verified cryptography.

2.4 WireGuard Coordination

In Section 2.3.3, the lack of a coordination entity on stand-alone WireGuard topologies was raised as a possible limitation. In fact, manually reconfiguring peers in a WireGuard network is a process that scales terribly. Moreover, on highly restrictive networks, creating direct WireGuard tunnels might prove to not be that easy.

With this in mind, this section explores applications and implementations of coordination platforms built, or with the potential to be built, on top of WireGuard, aiming to create a seamless peer orchestration and configuration process, minimizing human intervention.

2.4.1 Tailscale

Tailscale is a VPN service operating with a Golang user-space WireGuard variant as its data plane [30].

Architecture

Tailscale's central entity, referred to as the **coordination server**, functions as a shared repository of peer information, used by clients to retrieve data regarding other nodes and establish on-demand P2P connections among each other.

This approach differs from traditional hub-and-spoke since the coordination server carries nearly no traffic - it only serves encryption keys and peer information. Tailscale's architecture provides the best of both worlds, benefiting from the advantages of a centralized entity facilitating dynamic configurations and peer discovery without bottlenecking its data exchanging performance.

In practical terms, a Tailscale client will store, in the coordination server, its own public key and where it can currently be found. Then, it downloads a list of public keys and addresses that have been stored on the server previously by other clients. With this information, the client node is able to configure its WireGuard interface and start communicating with any other node in its domain.

Overcoming restrictive networks

There are however, some network security configurations which prevent WireGuard tunnels from being established. For example, firewalls blocking User Datagram Protocol (UDP) traffic entirely prevent the creation of direct WireGuard tunnels between two peers (as WireGuard is a UDP only protocol). In addition to orchestrating WireGuard peers, Tailscale is also able to overcome such constrained networks.

Regarding stateful firewalls, where, generally, inbound traffic for a given source address on a non-open port is only accepted if the firewall has recently seen an outgoing packet with such *ip:port* as destination (essentially assuming that, if outbound traffic flowed to a destination, the source expects to receive an answer from that same destination), Tailscale keeps, in its coordination server, the *ip:port* of each node in its network. With this information, if both peers send an outgoing packet to each other at approximately the same time (a time delta inferior to the firewall's cache expiration), then the firewalls at each end will be expecting the reception of packets from the opposite peer. Hence, packets can flow bidirectionally and a P2P communication is established. To ensure this synchronism of attempting communication at approximate times, Tailscale uses its coordination server and Designated Encrypted Relay for Packets (DERP) servers (explored further in the following paragraphs) as a side channel.

Although this procedure is quite effective, in networks with NAT mechanisms, where source and destination addresses are tampered with, this process is not as straightforward, since peers don't know the public addresses NAT will translate their private addresses to. The Session Traversal Utilities for NAT (STUN) protocol offers aid in performing NAT-traversal operations [13] and can solve this problem. For a peer to discover and store in the coordination server its own public *ip:port*, it first sends an UDP packet to a STUN server. Upon receiving this packet, the STUN server can see which public source address was used (the address NAT translated to) and replies this value to the peer.

There are, however, some NAT devices that create a completely different public address mapping for each different destination a machine communicates with, which hinders the above address discovery process. Such devices are classified as Endpoint-Dependent Mapping (EDM) (in opposition to Endpoint-Independent Mapping (EIM)) [31].

Networks employing EDM devices and/or really strict firewall rules, render these traversal techniques useless. To enable P2P communications in such scenarios, Tailscale also provides a network of DERP servers, which are responsible for relaying packets over Hyper-Text Transfer Protocol (HTTP). A Tailscale client is able to forward its encrypted packets to one of such DERP servers. Since a client's private key never actually leaves its node, DERP servers can't decrypt the traffic being relayed, performing only redirection of already encrypted packets. These relay servers are distributed geographically. However, traffic relayed by DERP servers always carry an increase in latency and loss of bandwidth, which isn't terrible, as the alternative is not being able to establish connections at all.

As such, Tailscale's design provides a set of directives and infrastructure that work together to ensure WireGuard tunnels can be set up between any two peers, regardless of what policies the network between them employs.

Direct vs Relayed Connections

Hence, Tailscale connections can be of two types: either **direct** or **relayed**. Tailscale will always attempt to create direct connections. In fact, if NAT-traversal succeeds on both endpoints, a direct WireGuard tunnel can be established. However, on networks employing EDM NAT mappings and/or UDP blocking firewalls, direct connections cannot be formed. That's where relayed connections are created. Clients on relayed connections encapsulate WireGuard UDP packets as Transmission Control Protocol (TCP) streams and forwards them through HTTP to the closest DERP server available, which in turn delivers them to its destination.

Headscale

While Tailscale's client is open source, its coordination server isn't. There is, however, an open-source, self-hosted alternative to Tailscale's control server, Headscale. Headscale [32] provides a narrow-scope implementation (with a single Tailscale private network) of the aforementioned control server, which, in the authors' words, is mostly suitable for personal use and small organizations. Nodes running Tailscale clients can opt to specify the location of the control server, which can be the address of a running self-hosted Headscale instance.

Headscale also supports the self-hosting of DERP and STUN servers, required when establishing relayed connections. This is useful as it allows the self-hosting of a complete Tailscale solution.

2.5 University of Aveiro Network

Due to security and privacy concerns, the specification of the UA's network topology is not publicly available nor is it made available for this dissertation. As such, it is perceived as a black box. There are, however, a few reachable conclusions derived from observing its behaviour.

First, the network is highly segmented, where clients are grouped according to their roles. In fact, when clients connect to an Access Point (AP) within the campus, they are connected to a Virtual Local Area Network (VLAN) shared by several other clients, in which the private resources are accessible. Then, the present NAT mechanisms in the network are unknown, as are their mappings' Time To Live (TTL), which is an important variable mentioned in Section 2.4.1 to perform NAT-traversal techniques. Finally, the network contains a segment for public services, which can be accessed from anywhere on the Internet. This public point of communication will allow clients from within the campus to connect to the control platform, serving in this domain.

Chapter 3

Methodology

Having outlined the relevant research, background and technologies for this dissertation's context, this chapter aims to present a work proposal for the solution implementation. The approach to be taken is segmented in three main stages: Prototype Development, Production Deployment and, finally, Automation.

3.1 Technology Stack

Based on the research done in the previous section, WireGuard seems the most adequate encrypted communication protocol for this project. However, to avoid the already covered added complexity of a stand-alone WireGuard network, Tailscale will function as the core of this solution. Ideally, all Tailscale related infrastructure should be self-hosted, which can be achieved using Headscale's open-source implementation of Tailscale's coordination server.

Therefore, our overlay networks are formed by a group of Tailscale clients, running in the robots, which are managed and coordinated by Headscale.

3.2 Prototype Development

This first phase aims to build a small-scale prototype in a virtual development environment. The prototype should, using Headscale as the coordination server and Tailscale as clients, be able to establish a P2P connection between two sample machines, which couldn't communicate beforehand. Hence, this phase's main goals are to (i) create a virtual environment which simulates the scenario being tackled, (ii) establish P2P connections between clients behind private NAT networks and (iii) validate the communication through the Robot Operating System (ROS) middleware. Obviously, regarding goal (i), the simulation of the networking conditions can't be entirely accurate, since, as referenced in section 2.5, details regarding UA's network mechanisms are vastly unknown. It is possible, however, to create an analogous situation, where clients can't immediately form P2P communications with each other but can all communicate with an external, public server. Moreover, as stated in previous sections, Tailscale's protocol efficiently deals with most network constraints preventing direct communication. In other words, it is only known clients can't communicate, the reasons behind why they can't are abstracted by Tailscale.

This virtual environment must be composed of three entities: a public server and two clients in their own private networks. As mentioned in section 2.4.1, Headscale is an open-source implementation of Tailscale's control server. Thus, the control server in this environment is a self-hosted Headscale instance deployed in the public server. Both clients, which can reach the public server, but not each other, should then authenticate in this control server and configure their Tailscale interfaces and addresses, allowing for a direct communication to take place.

Such an environment can easily be established in a single host, using virtualization software, with the use of three Linux virtual machines running on minimum resources. To achieve the networking requirements, as stated in goal (i), the client machines should be attached to individual private NAT Networks and connected via Wi-Fi to an AP, while the server machine should be attached to a bridge on the host's Ethernet interface. Figure 3.1 depicts this architecture.

Finally, to validate goal (iii), a simple ROS application should be deployed and tested in the clients, in which communication is done through the Tailscale interfaces.

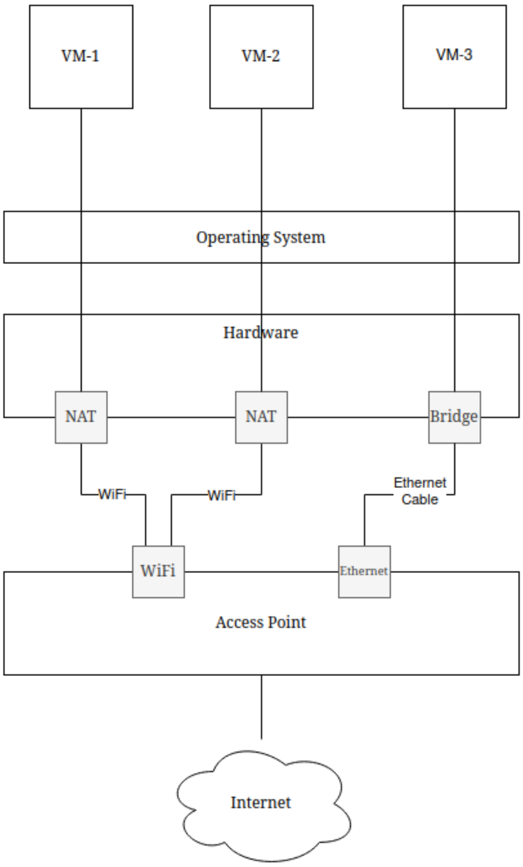

Figure 3.1: Development Environment Architecture

3.3 Deployment

After achieving and configuring the prototype previously developed, the next phase focuses on the deployment of the solution within UA's premises. As already mentioned, the Headscale instance should be hosted in a server accessible from anywhere within the campus. Here, the configuration should be based on the research done during the previous phase, with few eventual changes to ports and/or addresses. Regarding clients, ideally, each team of robots should have its own user, which individual robots will use for authentication. ¡TODO: COMO SERA O PROCESSO DE GERAR KEYS???¿.

Hence, this phase aims to (i) deploy an Headscale server available to any client connected to UA's network, (ii) configure robots to act as Tailscale clients and (iii) establish secure communication between a team of robots spread out throughout the campus.

3.4 Validation

(i) Measure network performance on normal vs Tailscale connections (ii) Validate connections spread out through various campus points (iii) Test the solution with demanding apps, such as live video streaming, through Tailscale interfaces.

3.5 Automation

This final development phase aims to create automation processes to aid in the deployment and configuration of the solution.

So, the automation process has two main goals in mind: First, to (i) provide a minimal configuration script to install and deploy an Headscale instance and provision its respective resources (auth keys, users, Access Control Lists (ACLs)) and (ii) provide a script installing Tailscale's client and configuring the system to automatically join the desired network.

3.6 Work Plan

The tasks encompassing the phases described above can be summed up in a Gantt diagram, presented in figure 3.2. The tasks to be carried out are as follows:

- **Prototype Development** - Implementation, in the development environment, of a prototype fulfilling the requirements. The end goal is to establish a P2P connection between the two machines that can't communicate.

- **Deployment** - Deployment of the control server in UA's public services domain and configuration of clients in the robots.

- **Validation** - Validation of the deployed solution, ensuring encrypted communication between nodes in different geographical locations within the campus.

- **Automation** - Development of configuration based scripts automating client and server configurations.

- **Final Document Writing** - Writing of the final document, presenting the development process and providing analysis of respective results.

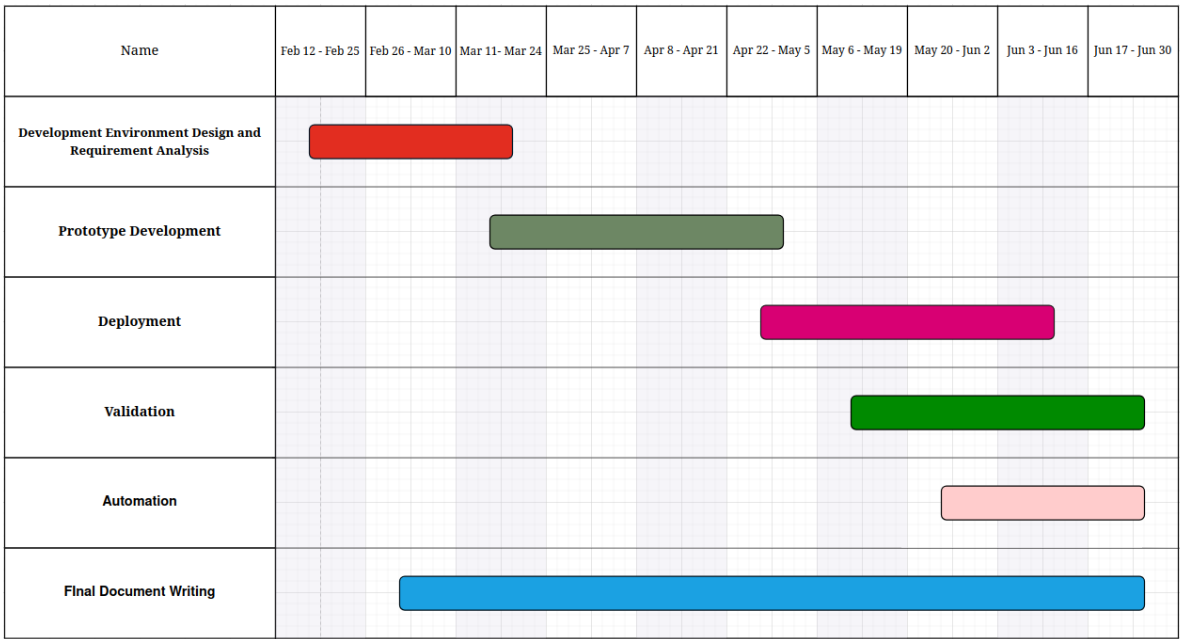

Figure 3.2: Development planning proposal

Chapter 4

Prototype Development

This chapter details the configuration of the development environment, as proposed in figure 3.1, and its use for the implementation of a narrow scope prototype. This environment is managed with Oracle Virtual Box. Having defined the goals for this experiment in the afore stated work proposal, this prototype requires the deployment of the Headscale control server, which is hosted in **VM-3**, followed by the configuration, and respective authentication of both clients, **VM-1** and **VM-2**, using the Tailscale Command Line Interface (CLI). With the clients configured with Tailscale addresses, these can be used to run tests on ROS applications.

4.1 Development Environment

4.1.1 Access Point

The access point used in this environment is a N600 Wireless Dual Band Gigabit router. The host machine is connected to the access point via Ethernet. With this setup, when one of the client machines wants to communicate with **VM-3**, the traffic will be routed from the client to the AP's Wi-Fi interface. Then the router will forward the packet through its Ethernet interface, reaching the host machine and, consequently, reaching **VM-3**, as its network adapter is bridged to the host's Ethernet interface.

4.1.2 Virtual Machines

The three virtual machines composing the environment run *Ubuntu Server 22.04* as their operating system. Due to the nature of the goals to be achieved in this phase, which focus on an extremely narrow scope, the machines require very little resources. Regarding networking, **VM-3** is attached to a bridge network on the Ethernet interface of the host machine. As for **VM-1** and **VM-2**, each respective network adapter is attached to a NAT, isolating each client on its own private network, inaccessible from the outside. All machines can access the internet through the access point, described in the previous section.

Both **VM-1** and **VM-2** are assigned the same private IP address since they are residing in distinct private networks. For convenience, the Virtual Machines are also configured to allow Secure Shell (SSH) connections, which means port 22 is open on all machines. Moreover, for machines **VM-1** and **VM-2**, which are confined to their private networks, this was achieved with port forwarding rules, forwarding **VM-1**'s port 22 to the host machine's port 2222 and **VM-2**'s port 22 to the host machine's port 2223.

	VM-1	VM-2	VM-3
Operating System	Ubuntu Server 22.04	Ubuntu Server 22.04	Ubuntu Server 22.04
Memory (Mb)	1024	1024	1024
Storage (Gb)	10	10	10
CPUs	1	1	1
Network Adapter	NAT	NAT	Bridged (ethernet)
Address	10.0.2.15	10.0.2.15	192.168.10.214

Table 4.1: Development Virtual Machines specification

Table 4.1 summarizes said specification.

4.2 Headscale Instance Deployment

Headscale provides an highly configurable open-source implementation of Tailscale's coordination server. At the time of writing, the latest stable Headscale release is *v0.22.3* [1], which is the version of the software referred to in the rest of this chapter.

The Headscale instance must be made available to all clients. In this environment, that means Headscale is running in **VM-3**, since the two other Virtual Machines can communicate with it directly. Hence, in this server, Headscale was downloaded and installed.

4.2.1 Configuring Headscale

Headscale includes in its software a configuration YAML used to tweak parameters of the coordination server. First, the **server_url** field, which dictates the address clients should connect to for authentication, points to **VM-3**'s IP address, 192.168.10.214, on port 8080. Regarding Tailscale IP assignments, the instance uses the default subnet prefixes, 100.64.0.0/10 for ipv4 and fd7a:115c:a1e0::/48 for ipv6. Registered clients will be assigned IP addresses in these ranges. For the scope of this environment, no other additional configurations are necessary.

4.2.2 Development Users

With the service running, **VM-3** is now listening for Tailscale clients to connect and start using the protocol. To authenticate, clients are required to login under a user. By default, Headscale does not create any users automatically. Hence, for development purposes, an Headscale user, **dev**, was created in the instance and shall be the user the clients will register themselves with.

4.2.3 Services

Table 4.2 presents the available services and their respective listening configuration produced by the running Headscale instance. Besides the metrics and gRPC services, which are not required by clients to use the Tailscale protocol, are listening privately, on the server's localhost interface. The remaining services are exposed in the defined ports, reachable by the clients.

[1] Headscale's official releases, hosted in GitHub. https://github.com/juanfont/headscale/releases.

Service	Listen Address	Port	Description
Control Server	192.168.10.214	8080	Main service implementing Tailscale's coordination mechanisms
Metrics	127.0.0.1	9090	Exposes the /metrics endpoints, for monitoring

Table 4.2: Services running in the Headscale instance

	VM-1	VM-2
Tailscale Hostname	dev-1	dev-2
Tailscale IP (v4)	100.64.0.1	100.64.0.2
Tailscale IP (v6)	fd7a:115c:a1e0::1	fd7a:115c:a1e0::2

Table 4.3: Clients Tailscale configuration after authentication

4.3 Client Configuration

Initially, clients can't really establish a direct connection in a traditional way. In fact, neither client is assigned a public address which could be used as a communication's endpoint, nor do they possess any information regarding one another. They can, however, reach the outside Internet, which consequently implies the translation of their private addresses into public ones, a process carried by the adapter attached to the host machine's NAT. ¡TODO: explicar teoria de porque é que o tailscale vai funcionar aqui¿

Using Tailscale's CLI, a client is able to authenticate in the Headscale instance previously deployed, which in turn configures its Tailscale interface with a respective Tailscale IP, in the range previously configured in the control server. This will allow a state where P2P WireGuard tunnels can be established freely between the registered clients.

Hence, the Tailscale binaries were installed in each client, using Tailscale's install shell script [2].

At this point, clients are ready to perform authentication in the control server and start communicating, a process described in the next section.

4.4 Authentication

Authenticating in the Headscale instance can be done either using a pre-authenticated key, generated by the control sever and shared with a client, or by accessing the instance through the browser in the client side. For automation purposes, the authentication keys provide a much more useful mechanism.

With that said, reusable keys were generated in the control server with Headscale's **headscale preauthkeys create**, an utility included in the software's CLI, and shared with its respective clients. Clients are now able to authenticate with Tailscale's CLI, by using the **tailscale up** command. Two additional command-line parameters are set, the *login-server*, which points to the address previously defined in Headscale's config, and the *authkey*, where the shared pre-authenticated key is injected.

With the clients authenticated, both devices are respectively assigned a Tailscale IP and hostname. Table 4.3 presents the clients' Tailscale configurations. At this state, clients can communicate directly via Tailscale interfaces.

[2] Tailscale's install script, publicly available online. https://tailscale.com/install.sh

4.5 Communication with ROS

With both clients up and communicating, our last validation in this environment aims to ensure the connections are compatible with the ROS middleware. Therefore, a very simple ROS scenario was deployed in the clients. As the scope for this experiment lies solely on validating communication through Tailscale in a ROS context, clients are only required to run a very basic ROS distribution, hence, a no-GUI package, **ros-noetic-ros-base** [3] was installed in both clients.

The experiment starts by configuring **VM-1** as a ROS Master, achieved with the **ros-core** command which automatically assigns the host as the new master, listening on port 11311, under the **ROS_HOSTNAME** dev-1, matching its Tailscale hostname for convenience. Then, **VM-2** should acknowledge **VM-1** as the ROS master. The **ROS_MASTER_URI** environment variable points to where the ROS Master is listening. Hence, **VM-2** sets this variable with **VM-1**'s ROS Uniform Resource Identifier (URI), which is composed by **VM-1**'s Tailscale hostname and the previously established ROS port.

VM-2 successfully configures its ROS ecosystem with **VM-1** as its master, effectively validating the use of Tailscale tunnels in conjunction with ROS' middleware.

4.6 Chapter Summary

This chapter details a successful implementation of a minimal configuration prototype which simulates, in an analogous environment, the interactions between a self-hosted Headscale control server and its clients. In fact, the configured environment presented a situation where two nodes, due to their network topology, couldn't immediately establish a P2P connection, but could both reach a third node. Headscale was then introduced in the public node as a self-hosted Tailscale control server, which, after authentication operations via pre-shared keys, made it possible to connect the client nodes through the Tailscale interfaces. Finally, this tunnel was tested when used along with ROS, operating as expected. The procedure taken in this phase meets the goals outlined in section 3.2.

[3] ROS-Base (Bare Bones). Basic ROS packaging, build and communication libraries. No GUI. Pulled from public repositories.

Chapter 5

Production Deployment

The following chapter presents the configuration and deployment of a solution capable of being used by any client in the campus. As such, and as mentioned in previous sections, this implies that the Headscale instance must be available regardless of a client's physical location within UA's premises.

5.1 Headscale Deployment

In this solution, the Headscale instance is hosted within IRIS-Lab's network. This, however, only allows communications from clients residing in the laboratory's private network. For clients to be able to reach the instance from any location in the campus, port forwarding rules were created on an exposed server in IRIS-Lab's network, available to any client inside UA's network. This server forwards traffic on ports TCP/8080, for the main Headscale service and the embedded DERP and UDP/3478, running the STUN server, to the Headscale server. With this configuration, clients within the campus, which can directly reach the public facing proxy machine, can communicate with the Headscale instance. Figure 5.1 presents such architecture.

Since Headscale's host is also an Ubuntu Linux environment, installation of this service followed the process described in the development environment (see Section 4.2). Regarding configuration, however, the **server_url** parameter, which points to the endpoint clients should use to authenticate, is now the Fully Qualified Domain Name (FQDN) of the proxy server, *iris-lab.ua.pt.*

5.1.1 Self-Hosting DERP Infrastructure

Initially, Headscale was configured to use Tailscale's DERP server fleet. However, upon testing this scenario, relayed connections implied the redirection of traffic to a DERP server in Germany, resulting in tunnels with an average latency of 452ms, a value unacceptable for the scope of this dissertation. Moreover, as this solution is meant to be used exclusively by UA's clients, traffic should be contained within the campus' network. Therefore, Tailscale's DERP fleet was discarded, opting for a self-hosted alternative.

Headscale allows the use of an embedded DERP server, which also exposes STUN functionalities for NAT-Traversal. This server can be enabled through Headscale's configuration file, and runs alongside Headscale's main service, on the same address and port. As for the

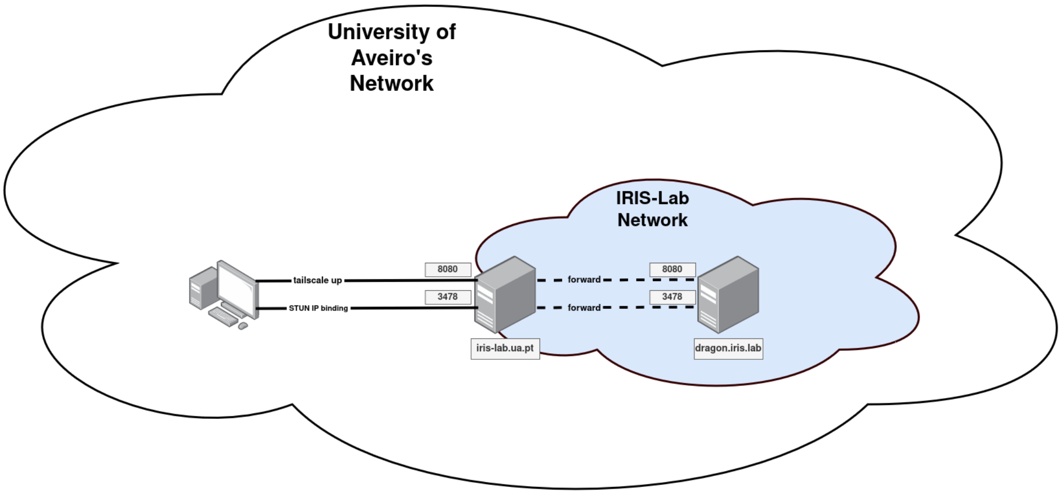

Figure 5.1: Production Deployment Architecture

STUN endpoints, these are also configured to run in the same address, on UDP port 3478.

Using the embedded self-hosted DERP allows relayed communication between clients to be much more efficient, with much more appealing latencies, discussed in Chapter 7.

5.1.2 Adding server certificates

To use the self-hosted DERP server, Headscale is required to be exposed on an Hyper-Text Transfer Protocol Secure (HTTPS) server, hence it needs a valid X509 certificate. Although Headscale's configuration allows a seamless integration with certification tools such as Let's Encrypt (LE) or Caddy, in this scenario, where Headscale's FQDN is not public, such methods aren't as straightforward. Alas, to certify Headscale, self-signed certificates were generated using the **openssl** CLI and added manually to the instance, by pointing to both certificate and key file paths through Headscale's configuration file. TODO: JUSTIFICAR PORQUE É QUE NAO HA ATTACK VECTOR.

Using self-signed certificates requires additional client certificate trust configurations, which shall be covered in the following section.

5.2 Client Deployment

5.2.1 Tailscale and Self-Signed Certificates

Tailscale's client does not support the use of self-signed certificates, even if the system trusts the certificate. Figure 5.2.1 presents the logic behind Tailscale's certificate validation. However, as an open-source codebase, this code can be freely modified and the binaries rebuilt to suit one's needs. Therefore, a fork of Tailscale was created, which shall be referred to as Selfscale [1].

[1] Selfscale's repository: https://github.com/VascoRegal/selfscale

```
conf.VerifyConnection = func(cs tls.ConnectionState) error {
    // Perform some health checks on this certificate before we do
    // any verification.
    if ht != nil {
        if certIsSelfSigned(cs.PeerCertificates[0]) {
            // Self-signed certs are never valid.
            ht.SetTLSConnectionError(
                cs.ServerName,
                fmt.Errorf(”certificate is self-signed”)
            )

        } else {
            // Ensure we clear any error state for
            // this ServerName.
            ht.SetTLSConnectionError(cs.ServerName, nil)
        }
    }
    ...
```

Figure 5.2: Snippet from Tailscale's client certificate verification method. Self-signed certificates are always considered invalid. /net/tlsdial/tlsdial.go, lines 79-90

Selfscale implements a new command-line option to the **tailscale** command, *–allow-self-signed*, which stores a boolean value. When true, the certificate validation method will not run the self-signed validation condition.

With the use of this option, a self-signed certificate needs only to be trusted by the system to be valid in the eyes of Tailscale's client. Hence, this solution requires clients to run the binaries generated from Selfscale's source code.

5.2.2 Trusting Headscale

As the Tailscale client can now accept self-signed certificates, Headscale's certificate can be added to the client machine's trusted list. On Debian distributions, this required moving the certificate to the */usr/local/share/ca-certificates* and updating the list with the command **update-ca-certificates**. Installation scripts are available on a public repository.

5.3 Chapter Summary

This chapter details how a production solution was achieved. Headscale was deployed on a server inside IRIS-Lab's private network. Since using Tailscale's DERP servers resulted in unacceptable latencies, this server also hosts an embedded DERP server and a STUN server. To make the Headscale instance available to any client within the campus while deployed in IRIS-Lab, forwarding rules were configured on an IRIS-Lab server exposed to UA's network clients.

To certificate Headscale and run it on HTTPS, required when using self-hosted DERP servers, self-signed certificates were generated, as the forwarding server is not exposed to the

Internet.

Regarding clients, instead of using the official Tailscale build, which does not allow the use of self-signed certificates, a custom version was built, Selfscale, implementing an option to handle this scenario. Finally, Headscale's certificate was added to the client's trust system. Hence, clients can login by accessing the FQDN of the forwarding server and using this new option.

Such deployment allows any two clients connected to UA's network to communicate directly using Tailscale interfaces from anywhere in the campus.

Chapter 6

Automation

This chapter details the development of automation tools to speed up deployment and configuration processes. In Section 3.5, two main goals were defined, related to automatic deployments of the control server and clients. First, automating the deployment of a configured Headscale instance. This implies automating software installation, configuration applying and certificate management. Regarding clients, deployments require the installation of the custom binaries, explored in Section 5.2.1, adding Headscale's certificate to the trust list and, finally, authenticate to a desired network.

6.1 Distributing Selfscale binaries

Tailscale's codebase provides a script to build the binaries, which is also used to build Selfscale. Additionally, a configuration file is provided to setup Tailscale's daemon as a service.

Regarding distribution, these are hosted as a compressed file in the GitHub repository. TODO: Distributing the software as a package would probably be more optimal.

To generate the compressed package, a shell script was developed, **build_selfscale.sh**, which builds both the tailscale and tailscale daemon binaries and zips them with the aforementioned configuration file. This zip is hosted on the **selfscale** folder of the repository. Currently this script is used to release a new version.

6.2 Headscale Deployment

Headscale's deployment requires both the installation of the software and its configuration, namely the configuration YAML and the server certificate. These last files are publicly available in the automation repository [1].

Hence, the **install_headscale.sh** shell script starts by downloading a desired Headscale package, where version and system architecture are specified with command-line arguments. Then, the script fetches the Headscale's configuration YAML, publicly available in the automation repository, and applies it to the instance. Finally, a x509 self-signed certificate is generated with the **openssl** CLI and placed, along with the certificate's key, in its respective path (matching what was defined in the the YAML).

[1] Project automation repository, publicly available. https://github.com/VascoRegal/ua-overlays-automation

6.3 Client Deployment

A shell script was also created for the clients. Here, configuring a client requires the installation of the modified Tailscale binaries, which can be downloaded as a zip from Selfscale's GitHub repository. Then, the package is extracted and files are placed accordingly. Finally, Headscale's server certificate is fetched and added to the system's trust mechanisms.

Running this simple script leaves a machine in a state where, given a pre-shared key, authentication can be performed with the **tailscale up** command, passing the additional *allow-self-signed* option.

6.4 Chapter Summary

This chapter covered how the processes involved in deploying and configuring both the Headscale instance and Tailscale clients were automated using shell scripts. These scripts create a minimal intervention procedure, useful when adding new clients to the overlays.

Chapter 7

Validation and Results

To validate the solution, two dimensions must be taken into account. First, it is necessary to ensure clients can use the overlay networks from anywhere within the campus. Then, the overall network performance must also be analysed, namely communication latencies, network TCP throughput, UDP packet loss and maximum bandwidth and network jitter. Fortunately, both scopes can be tested simultaneously! **iperf3** is an open-source tool which collects such metrics by exchanging packets on a client-server model.

Therefore, these tests were implemented as a python script [1] which connects to a host to run **iperf3** TCP and UDP tests for 120 seconds each and exporting the results as JavaScript Object Notation (JSON). **iperf3**'s TCP tests collect TCP network throughput and minimum, maximum and mean RTT time. UDP tests collect UDP throughput and packet loss.

For this experiment, a Tailscale client running **iperf3** as a server was placed inside IRIS-Lab's network. Then, with another machine, the script was run in clients connected to UA's network, from disperse points in the campus. This procedure allows not only the collection of **iperf3**'s network metrics and latencies but also the validation of the solution's coverage, since the script connects to the IRIS-Lab's host with its Tailscale IP. If this connection fails, Tailscale can't cover that area, possibly due to more constraining networking rules. As it will be shown in the following section, however, none of the tested areas failed to establish communication.

Twelve different test scenarios were run, presented on table 7.1. Since the **iperf3** server is running on IRIS-Lab network, tests can be executed either using the server's Wi-Fi interface or its Tailscale interface. Interpolation of the data from these two scenarios allows the visualization of Tailscale's overhead on communication. The remaining scenarios are meant to both ensure Tailscale can establish connections from any point in the campus, while also providing insights on general communication performance.

7.1 Campus Coverage

As mentioned above, all identified locations were able to communicate with the client running on IRIS-Lab through its tunnel interface, validating the solution's ability to establish Tailscale tunnels from anywhere within the campus.

[1] https://github.com/VascoRegal/ua-overlays-automation/blob/main/tests/network/iperf/analysis.py

Location	**Building ID**	**Connection Type**
IRIS-Lab (IRIS)	19	Internal
IRIS-Lab (IRIS)	19	Tailscale (direct)
University Library (BIB)	17	Tailscale (relayed)
Political Science Department (DSCPT)	12	Tailscale (relayed)
Communication and Arts Department (DECA)	21	Tailscale (relayed)
Student's Bar (BE)	N	Tailscale (relayed)
Eletronics, Telecomunications and Informatics Department (DETI)	4	Tailscale (relayed)
Exhibition Hall (EXPO)	24	Tailscale (relayed)
Pedagogical Complex (CP)	23	Tailscale (relayed)
Mathematics Department (DMAT)	11	Tailscale (relayed)
Biology Department (DBIO)	8	Tailscale (relayed)
Economics and Management Department (DEGEIT)	10	Tailscale (relayed)

Table 7.1: Locations and scenarios used for validation

7.2 Network Performance

7.2.1 Throughput

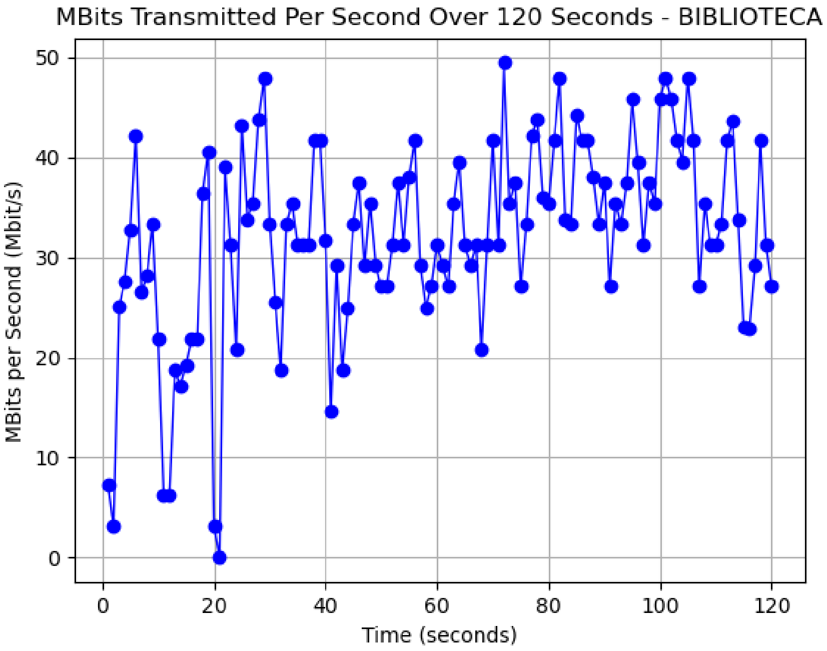

Figure 7.1: TCP Tailscale tunnel Network TCP Throughput (University Library)

7.2.2 Latency

7.2.3 Jitter

7.2.4 Packet Loss

Packet loss is a metric associated with UDP tests, included in **iperf3**. All UDP tests resulted in a 0 % packet loss.

7.2.5 Protocol Overhead

To evaluate Tailscale's overhead on communication, throughput and latency results via internal network connections were compared with the same scenario via Tailscale tunnel. For this test, both client and server were residing inside IRIS-Lab. Figure 7.5 presents the TCP throughputs results and figure 7.6 shows the average latencies. As expected, communicating through Tailscale implies a loss of throughput and a slight increase in latencies.

TCP Throughput Loss

Throughput loss can be calculated by the expression:

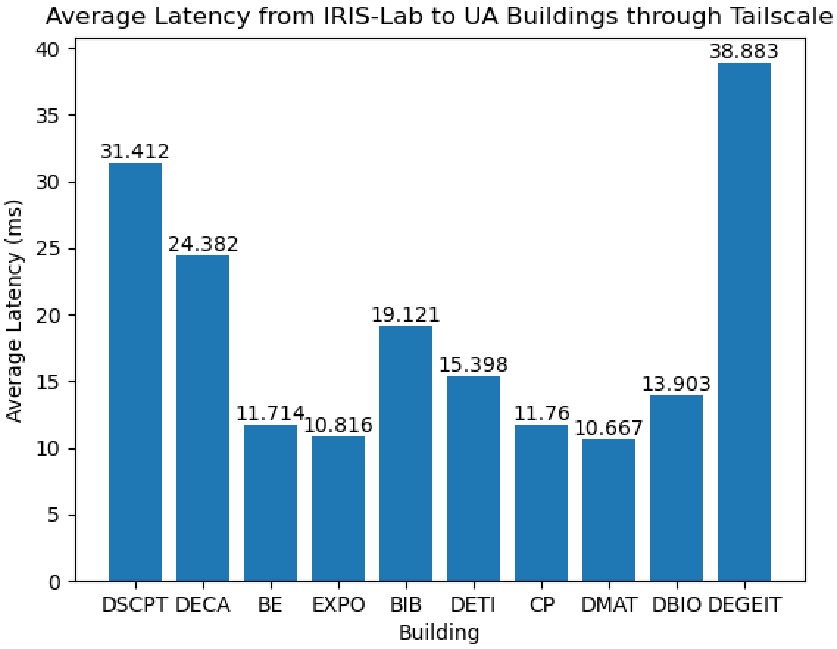

Figure 7.2: Average latencies by campus location

$$\text{Throughput Loss (\%)} = \left(\frac{\overline{\text{TP}}_I - \overline{\text{TP}}_T}{\overline{\text{TP}}_I} \right) \times 100$$

Where,

- $\overline{\text{TP}}_I$ is the average throughput for the internal connection.
- $\overline{\text{TP}}_T$ is the average throughput for the Tailscale connection.

Which results in a throughput loss of **39.30** %

Latency Increase

Latency increase is measured with a similar approach, calculated with:

$$\text{RTT Increase (\%)} = \left(\frac{\overline{\text{RTT}}_T - \overline{\text{RTT}}_I}{\overline{\text{RTT}}_I} \right) \times 100$$

Where,

- $\overline{\text{RTT}}_I$ is the average RTT for the internal connection.
- $\overline{\text{RTT}}_T$ is the average RTT for the Tailscale connection.

Resulting in a latency increase of **35.25** %

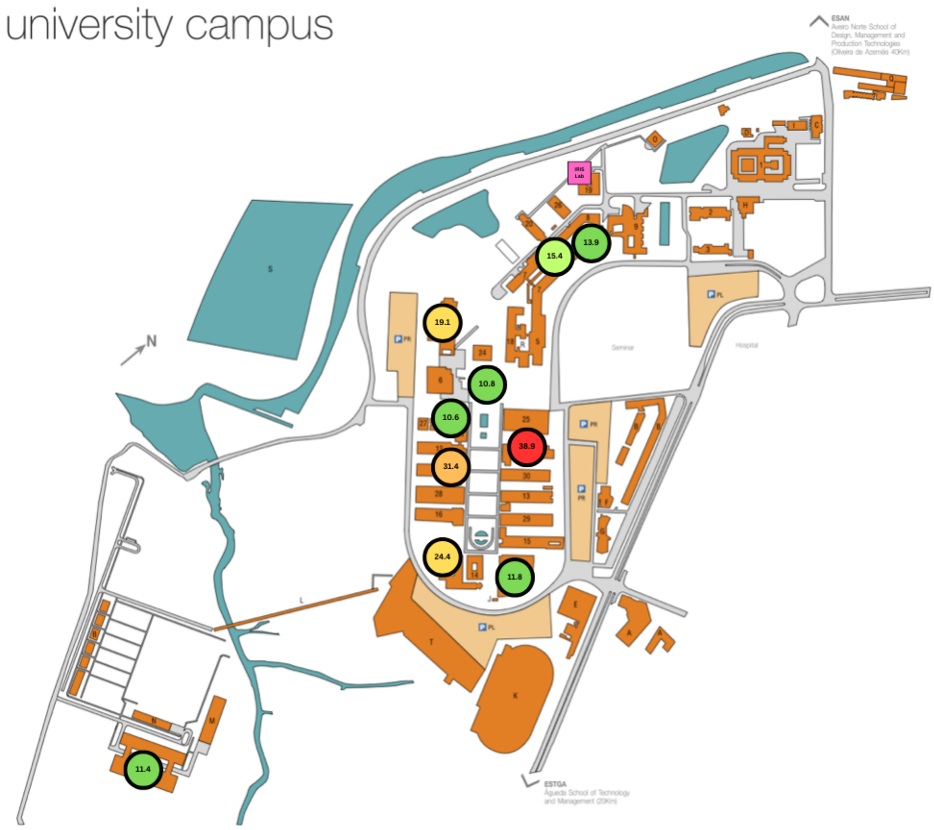
university campus
ESAN
Aveiro Norte School of Design, Management and Production Technologies (Oliveira de Azeméis 40Km)
IRIS Lab
N
15.4
13.9
19.1
10.8
10.6
38.9
31.4
24.4
11.8
11.4
Seminar
Hospital
ESTGA
Águeda School of Technology and Management (20Km)

Figure 7.3: Average latency on a Tailscale connection from different UA buildings to IRIS-Lab. The numbers in the circles represent average latency values, in ms.

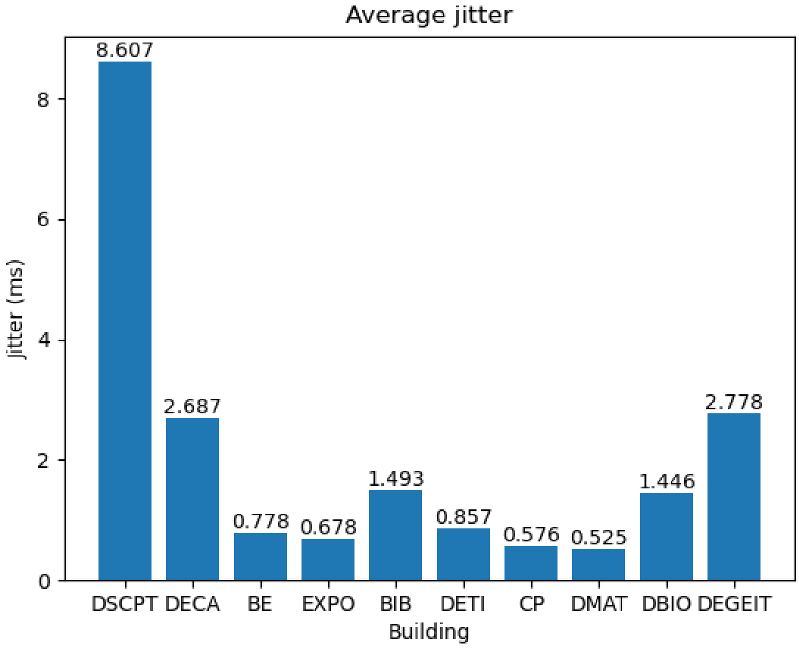

Figure 7.4: Network Jitter by campus location.

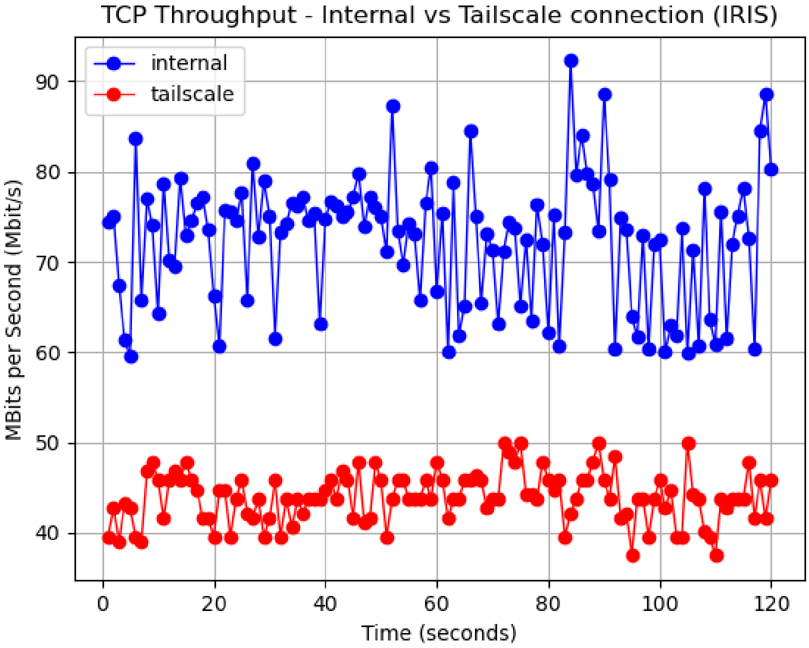

Figure 7.5: Tailscale's toll on throughput, tested on internal and Tailscale connections inside IRIS-Lab

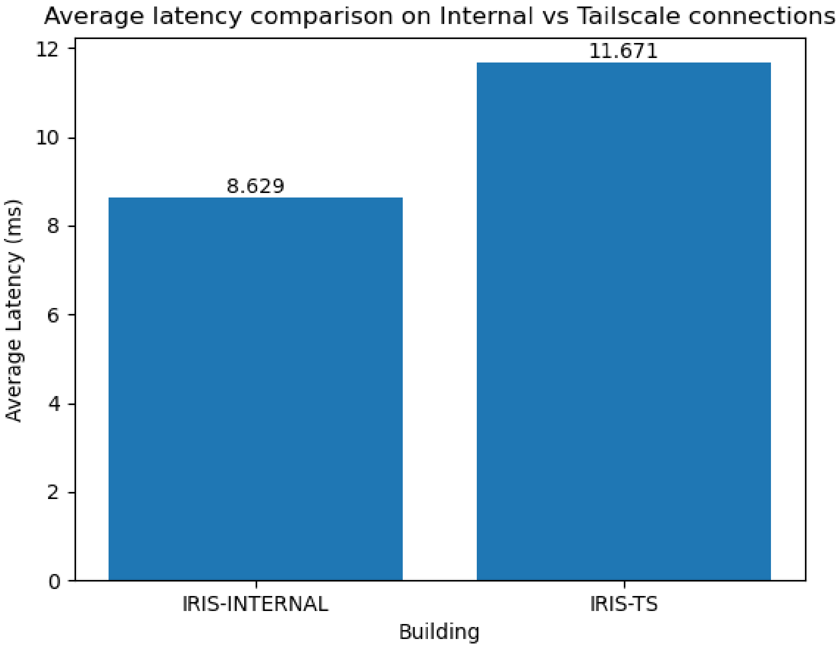

Figure 7.6: Average latency on internal and Tailscale connections

Chapter 8

Future Work

Chapter 9

Conclusion

Bibliography

[1] Peterson, Larry L. and Davie, Bruce S., *Computer Networks, Fifth Edition: A Systems Approach.* Morgan Kaufmann Publishers Inc., 2011.

[2] Andersen, D.G. and Balakrishnan, H. and Kaashoek, M.F. and Morris, R., "The case for resilient overlay networks," in *Proceedings Eighth Workshop on Hot Topics in Operating Systems*, 2001.

[3] S. Troia, L. M. M. Zorello, A. J. Maralit, and G. Maier, "Sd-wan: An open-source implementation for enterprise networking services," in *2020 22nd International Conference on Transparent Optical Networks (ICTON)*, 2020.

[4] S. Troia, L. M. Moreira Zorello, and G. Maier, "Sd-wan: how the control of the network can be shifted from core to edge," in *2021 International Conference on Optical Network Design and Modeling (ONDM)*, 2021.

[5] C. Zhang, P. Dhungel, D. Wu, and K. W. Ross, "Unraveling the bittorrent ecosystem," *IEEE Transactions on Parallel and Distributed Systems*, 2011.

[6] J. T. Harmening, "Chapter 58 - virtual private networks," in *Computer and Information Security Handbook (Third Edition)*, Morgan Kaufmann.

[7] André Zúquete, *Segurança em Redes Informáticas.* FCA, 2013.

[8] T. Berger, "Analysis of current vpn technologies," in *First International Conference on Availability, Reliability and Security (ARES'06)*, 2006.

[9] S. Elhedhli and F. X. Hu, "Hub-and-spoke network design with congestion," *Computers Operations Research*, 2005.

[10] M. E. O'Kelly, "A geographer's analysis of hub-and-spoke networks," *Journal of transport Geography*, 1998.

[11] Y. An, Y. Zhang, and B. Zeng, "The reliable hub-and-spoke design problem: Models and algorithms," *Transportation Research Part B: Methodological*, 2015.

[12] A. Keränen, C. Holmberg, and J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal." RFC 8445, 2018.

[13] M. Petit-Huguenin, G. Salgueiro, J. Rosenberg, D. Wing, R. Mahy, and P. Matthews, "Session Traversal Utilities for NAT (STUN)." RFC 8489, 2020.

[14] K. Seo and S. Kent, "Security Architecture for the Internet Protocol." RFC 4301, 2005.

[15] C. Kaufman, P. E. Hoffman, Y. Nir, P. Eronen, and T. Kivinen, “Internet Key Exchange Protocol Version 2 (IKEv2).” RFC 7296, 2014.

[16] S. Kent, “IP Authentication Header.” RFC 4302, 2005.

[17] S. Frankel, K. Kent, R. Lewkowski, A. D. Orebaugh, R. W. Ritchey, and S. R. Sharma, “Guide to ipsec vpns:.,” 2005.

[18] T. S. Nam, H. Van Thuc, and N. Van Long, “A High-Throughput Hardware Implementation of NAT Traversal For IPSEC VPN,” *International Journal of Communication Networks and Information Security*, 2022.

[19] C. Singh and K. Bansal, “NAT Traversal Capability and Keep-Alive Functionality with IPSec in IKEv2 Implementation,” 2012.

[20] J. Yonan, “Openvpn.” `https://openvpn.net/`.

[21] J. A. Donenfeld, “Wireguard: next generation kernel network tunnel.,” in *NDSS*, 2017.

[22] J. Ćurguz *et al.*, “Vulnerabilities of the ssl/tls protocol,” *Computer Science & Information Technology*, 2016.

[23] D. J. Bernstein, “Curve25519: new diffie-hellman speed records,” in *Public Key Cryptography-PKC 2006: 9th International Conference on Theory and Practice in Public-Key Cryptography, New York, NY, USA, April 24-26, 2006. Proceedings 9*, 2006.

[24] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, “Hacl*: A verified modern cryptographic library,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[25] D. J. Bernstein *et al.*, “Chacha, a variant of salsa20,” in *Workshop record of SASC*, 2008.

[26] G. Procter, “A security analysis of the composition of chacha20 and poly1305,” 2014.

[27] T. Perrin, “The noise protocol framework,” 2018.

[28] S. Mackey, I. Mihov, A. Nosenko, F. Vega, and Y. Cheng, “A performance comparison of WireGuard and OpenVPN,” in *Proceedings of the Tenth ACM Conference on data and application security and privacy*, 2020.

[29] L. Osswald, M. Haeberle, and M. Menth, “Performance comparison of vpn solutions,” 2020.

[30] A. Pennarun, “How tailscale works.” `https://tailscale.com/blog/how-tailscale-works/`, 2020.

[31] C. F. Jennings and F. Audet, “Network Address Translation (NAT) Behavioral Requirements for Unicast UDP.” RFC 4787, 2007.

[32] J. Font and K. Dalby, “Headscale.” `https://headscale.net/`, 2023.