Politecnico di Torino

Cybersecurity for Embedded Systems
01UDNOV

Master's Degree in Computer Engineering

Implementation of a VPN client for FreeRTOS

Project Report

Candidates:
Lorenzo Chiola (287911)
Simone Pistilli (287607)
Francesco Spagnoletti (291079)
Lucia Vencato (292614)

Referents:
Prof. Paolo Prinetto
Dr. Matteo Fornero
Dr. Vahid Eftekhari

Contents

List of Figures

Abstract

When working on any IoT (Internet of Things) project, security should always be a top priority, this because the more devices are connected to the internet, the more attractive the data becomes for cybercriminals. All of these embedded systems deal with sensitive data that can be harmful if it falls into the wrong hands. Even the smallest Internet of Things system can reveal a lot about the real world in which these devices operate, and they may even be able to access and control it themselves. As a result, may IoT projects have begun to implement security measures. Implementing a Virtual Private Network (VPN) is one of the most popular solutions. Through this system users can send and receive data across public networks as if their computers were directly connected to a private network, knowing that all of their data is encrypted and their IP address is hidden. For this purpose, this paper suggests to use WireGuard as a VPN and see if it is a viable security solution for an IoT project. WireGuard uses state-of-the-art cryptography, which makes it faster, more secure, and more friendly to mobile and IoT devices than other protocols like OpenVPN or IPsec. To test this solution, an ESP32 was chosen as the development board, with FreeRTOS as the Real Time Operating System, and a TCP/IP communication was established.

CHAPTER 1

Introduction

The Internet of Things (IoT) refers to the billions of physical devices connected to the internet around the world, all of which collect and share data. For this reason people should consider using more sophisticated security to ensure the data they collect is secure. Nowadays, everyone can control lights from outside the house, control thermostats with their phones, and order from online stores simply by speaking out loud. One thing that may go unnoticed is that all of this data is very personal and unique to the individual. These details may appear insignificant on their own. The reality is that these various devices are logging information on shopping habits, location, and even passing conversations and these devices can be vulnerable to all sorts of weaknesses. The issue arises when people intercept that data, make a copy of it, and then use it for their own purposes.
That is where the VPN comes in.
A VPN is a Virtual Private Network that encrypts all data sent between a device and a server. Furthermore, it obscures the device's IP address, making it impossible to determine from where the data was sent. For this reason, many companies that allow employees to work remotely will require them to connect to the company's business network via a VPN; sensitive data always takes a secure route to or from the local area network. If someone decides to use this service to protect their data, it is possible to search a VPN provider, but this solution has a cost, even if it is often reasonable. On the other side, a free implementation will undoubtedly have less support and may be more difficult to use for the average user.
Furthermore, it is difficult to implement a VPN on an IoT device because they require a large amount of computational resources. WireGuard as a VPN is becoming increasingly popular because it is simple, employs cutting-edge cryptography, and is also free and open source.

The goal of the project described in this report is to implement a VPN client for FreeRTOS in order to establish a secure communication channel between a client and a server that allows for confidentiality, integrity, and authentication.
The choice settled on WireGuard protocol, because of its lightness and simplicity.
The ESP32 was chosen as development platform because modules are affordable and provide all of the required functionalities, particularly the WiFi module, the LwIP network stack and the true random number generator. Moreover the company of the ESP32, Espressif, offers a development framework called ESP-IDF which provides a slightly modified version of FreeRTOS.

This paper is divided into several chapters. The second one provides a brief overview of all the topics required to follow the project's development. The third chapter provides a general overview of project implementation for the ESP32, while the fourth exposes all technical details. The fifth chapter focuses on a similar development, but in a more general scenario, using the FreeRTOS simulator for Linux. The sixth chapter summarises all of the findings of the research, as well as the issues encountered and potential future projects. The report is concluded in the seventh chapter. Appendix

A contains a detailed user manual for executing, step-by-step, the projects that were developed.

CHAPTER 2

Background

This chapter provides a summary of all of the project's topics. Starting with the main one, what is a VPN and specifically how WireGuard works, followed by an overview of the operating system used, FreeRTOS. There is also a description of the Lightweight IP Stack and of the board used for the project.

2.1 VPN

A VPN is a Virtual Private Network that guarantees confidentiality, integrity and authentication, through a secure communication channel (VPN tunnel).
With this service all the Internet traffic is encrypted and the user is able to protect his online identity, in fact it allows to mask the IP (Internet Protocol) and so the real position.
The term *Virtual* means that the devices in this network can be located anywhere in the world, and it is not necessary that they are all under the same local network.

2.1.1 VPN classification

There are two types of VPN:

- Remote Access VPN: it allows the user to connect to a server on a private network.
- Site-To-Site VPN: it refers to a connection set up between multiple networks which allows secure routing and communication.
  Considering the level of reliability and safety, this type of VPN can also be divided in:
  - Trusted: Internet Service Provider (ISP) guarantees the data protection.
  - Secure: it uses encryption protocols to guarantees the creation of a secure tunnel between the private network nodes. In this way the data inside the tunnel are protected against possible external attacks.
  - Hybrid: it is a mix of a secure VPN and a trusted one.

2.1.2 How VPN works

A VPN uses a tunnelling mechanism, that allows the creation of a secure channel between two remote entities. The data packets sent in this channel are encapsulated by the tunneling protocol and encrypted.
In this way the data are protected against possible external attacks and all the traffic is *invisible* in

the public network, because the user ip is masked so formally he is not receiving or sending anything. In the image 2.1 it can be seen the logical tunnel that protects the data packets.

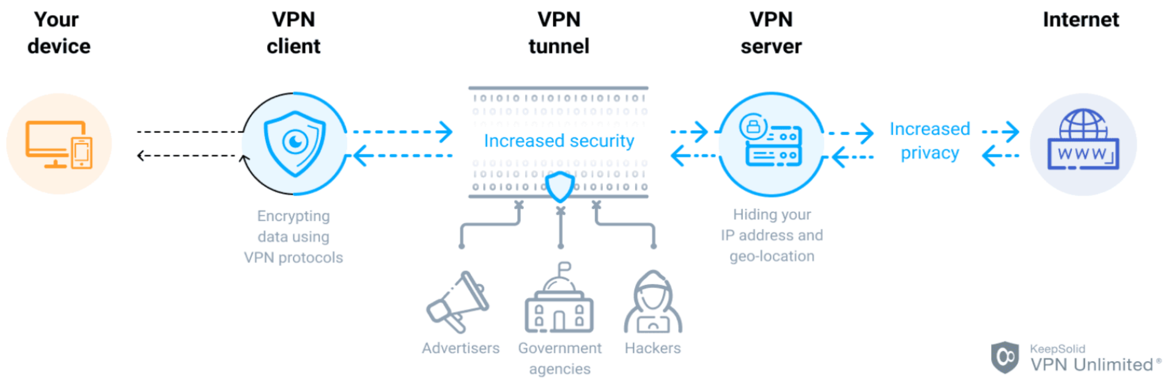


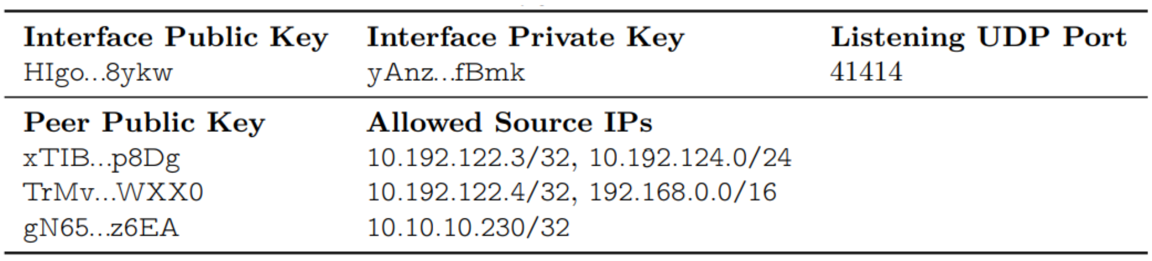
Figure 2.1: VPN tunnelling

2.2 WireGuard

Among all the available VPN, Wireguard [1] was the one chosen for this project.
It is an extremely simple modern VPN which uses state-of-the-art cryptography and it is also Free and Open Source. WireGuard is a secure VPN, so through tunnelling, it securely encapsulates IP packets over UDP.
How it works is simple: it simply gives a virtual interface which can then be administered using the standard `ip(8)` and `ifconfig(8)` utilities. After configuring the interface with a private key and the various public keys of peers with whom it will communicate securely, the tunnel simply works.

2.2.1 How WireGuard works

The fundamental principle of a secure VPN is an association between peers and the IP addresses each is allowed to use as source IPs. In WireGuard, peers are identified strictly by their public key, so there is a simple association mapping between public keys and a set of allowed IP addresses.
In the following there is an example of a possible crypto key routing table 2.2:



Interface Public Key	**Interface Private Key**	**Listening UDP Port**
HIgo...8ykw	yAnz...fBmk	41414
Peer Public Key	**Allowed Source IPs**	
xTIB...p8Dg	10.192.122.3/32, 10.192.124.0/24	
TrMv...WXX0	10.192.122.4/32, 192.168.0.0/16	
gN65...z6EA	10.10.10.230/32	

Figure 2.2: Cryptokey routing table for a WireGuard Network Interface

The interface itself has a private key and a UDP port for listening, followed by a list of peers. Each peer is identified by its public key and each has a list of allowed source IPs.
It is important that peers are able to send encrypted WireGuard UDP packets to each other at

particular Internet endpoints. Each peer in the cryptokey routing table may optionally pre-specify a known external IP address and UDP port of that peer's endpoint (adding in 2.2 an Internet Endpoint in the form IP:UDPport). It is optional because if it is not specified and WireGuard receives a correctly authenticated packet from a peer, it will use the outer external source IP address for determining the endpoint.
When an outgoing packet is being transmitted on a WireGuard network interface, called wg0, this table is consulted to determine which public key to use for encryption.
Using 2.2, when receiving and sending a packet on interface wg0 there will be the following flow:
A packet is locally generated and is ready to be transmitted on the outgoing interface wg0:

1. The plaintext packet reaches the WireGuard interface, wg0.
2. The destination IP address of the packet, 192.168.87.21, is inspected, which matches the peer TrMv...WXX0
3. The symmetric sending encryption key and nonce counter of the secure session associated with peer TrMv...WXX0 are used to encrypt the plaintext packet using ChaCha20Poly1305.
4. A header containing various fields is prepended to the now encrypted packet.
5. This header and encrypted packet, together, are sent as a UDP packet to the Internet UDP/IP endpoint associated with peer TrMv...WXX0, resulting in an outer UDP/IP packet containing as its payload a header and encrypted inner-packet.

A UDP/IP packet reaches UDP port 41414 of the host, which is the listening UDP port of interface wg0:

1. A UDP/IP packet containing a particular header and an encrypted payload is received on the correct port.
2. Using the header WireGuard determines that it is associated with peer TrMv...WXX0's secure session, checks the validity of the message counter, and attempts to authenticate and decrypt it using the secure session's receiving symmetric key. If it cannot determine a peer or if authentication fails, the packet is dropped.
3. Since the packet has authenticated correctly, the source IP of the outer UDP/IP packet is used to update the endpoint for peer TrMv...WXX0.
4. Once the packet payload is decrypted, the interface has a plaintext packet. If this is not an IP packet, it is dropped. Otherwise, WireGuard checks to see if the source IP address of the plaintext inner-packet routes correspondingly in the crypto key routing table.
5. If the plaintext packet has not been dropped, it is inserted into the receive queue of the wg0 interface.

NOTE: This explanation part is provided as it reported on the official WireGuard whitepaper [1].

2.2.2 Cryptographic Algorithms

WireGuard uses state-of-the-art cryptography. Among all the possible algorithms, the ones used in this project are:

- BLAKE2S : it is a cryptographic hash function optimized for 8 to 32-bit platforms and it produces digests of any size between 8 bits and 256 bits.

- X25519 : it is an elliptic curve DiffieHellman key exchange using Curve25519, which is an elliptic curve offering 128 bits of security (256 bits of key size). It allows two parties to jointly agree on a shared secret using a non-secure channel.
- CHACHA20-POLY1305 : it is an Authenticated Encryption with Additional Data (AEAD) algorithm. It combines the ChaCha20 stream cipher (whose input includes a 256-bit key, a 32-bit counter, a 96-bit nonce and plain text) with the Poly1305 message authentication code.

Authenticated Encryption with Additional Data Algorithm (AEAD)

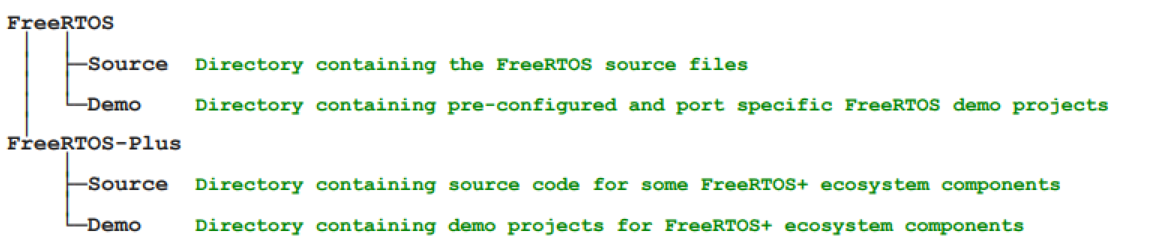
It is an algorithm that guarantees both confidentiality (through encryption) as well as integrity and authenticity of data.
Encryption only provides confidentiality, but the message sent is not protected against modification. So, additional data must be transmitted along with the message to authenticate it. For AEAD this operation takes the form of a MAC (Message Authentication Code).
Keyed hash functions are commonly used to generate MACs.

2.3 FreeRTOS

FreeRTOS is a real-time OS which is based on a simple kernel, and it is widely used in many embedded systems applications. It is owned and developed by Real Time Engineers Ltd. The application could be organized in different threads that could be executed on different type of processors and the execution order is computed by the kernel basing on the priority of the different tasks, which is assigned by the application designer. Usually, the tasks with more strict time requirements have the higher priority with respect to the one more time tolerant.

2.3.1 FreeRTOS distribution

FreeRTOS could be seen as a library composed of many C source files that provides multi-tasking capabilities. Compiling these files with the target application make it able to reach the FreeRTOS API. In order to simplify the development with this library, many different demo projects, that are preconfigured to compile the correct source files, are inserted in the FreeRTOS released folder.
FreeRTOS is configured by the FreeRTOSConfig.h header which is used to setup the OS to fit the target application. It contains different configuration in order to set parameters like, for example, the scheduling algorithm. This file must be located in a directory that is part of the application that has been built. The official distribution is given with first and second directory as can be seen in figure 2.3



```
FreeRTOS
 │   ├─Source   Directory containing the FreeRTOS source files
 │   └─Demo     Directory containing pre-configured and port specific FreeRTOS demo projects
 │
FreeRTOS-Plus
     ├─Source   Directory containing source code for some FreeRTOS+ ecosystem components
     └─Demo     Directory containing demo projects for FreeRTOS+ ecosystem components
```

Figure 2.3: FreeRTOS directory tree.

For more details on FreeRTOS you can check the official site ([2]) and the guide ([3])

2.4 The Lightweight IP Stack - LwIP

The Lightweight IP Stack (LwIP) is a very small complete implementation of many network protocols, most importantly it provides everything that is needed for a TCP connection (IP, ICMP, ARP), it is predisposed for some useful protocols like NTP, and PPP.
LwIP is built to be easy to port to new platform, which is needed for any hardware change.
Only a few files have to be written:

- `cc.h`: definition of basic types (for example u32) for the compiler in use (typically unsigned long or uint32_t)
- `sys_arch.c/.h`: functions needed for threaded operation, mainly `sys_now()` which returns the current time, and implementations of threads, semaphores and signals that use the underlying resources, usually a Real Time Operating System (RTOS).
- `perf.c/.h`: performance measurement macros (optional).
- netif (at least one): a netif is an LwIP network interface. It is described by a `struct netif`. The netif must implement at least:
  - an initialization function, which will be passed to `netif_add()` by the application when creating the interface.
  - some means to call the LwIP input call-back (`struct netif->input`) when data is received. Usually a thread function started during initialization, or a function to be called periodically by the application. The input call-back is set by `netif_add()`, it must be `tcpip_input()` for threaded systems, it can be `ethernet_input()` or `ip_input()` for `NO_SYS=1` ports, respectively if LwIP should implement the media access layer (only for ethernet links) or just the network layer (for any other interface).
  - an output function, to be called by LwIP to transmit data. A pointer should be saved in `struct netif->output`.
- `lwipopts.h`: LwIP settings, anything that is not defined will take a default value from opt.h.

Some example ports are provided in the auxiliary distribution called "contrib" (which indicates code contributed from developers outside the project).

LwIP can be ported to a platform that already has a Real Time Operating System, in which case all the facilities for the port already exist in the RTOS, so writing `sys_arch.c` is easiest. Care should be taken to ensure that the scheduler is already running when performing operations that use the network hardware, since transmission and reception employ LwIP threads (which are implemented as RTOS tasks). The basic procedure is available at [4].

LwIP can also be ported to bare-metal platforms that don't have an Operating System. "`#define NO_SYS 1`" [5] must be added to `lwipopts.h`, and some library functions must be called periodically by user code to poll the network interfaces and other parts of the stack. Threads and other OS features will not be used, so the netconn and socket APIs that need those will not be available to the application. For more information read [6].

There are three APIs available for applications:

- Raw / Callback API: [7] the user registers call-backs for events (such as reception of one packet). Also used for LwIP internals and for modules, like the WireGuard VPN module, that have to be portable also to NO_SYS platforms.
- netconn API: [8] available only in threaded ports of LwIP. Every feature is accessed through functions named `netconn_*()`, some of which block execution while waiting for data or events.

- Berkeley / BSD Socket API: [9] available only in threaded ports of LwIP. Modeled after the networking APIs of UNIX operating systems, with functions like `socket()` and `connect()`. The simple TCP examples written for this project use this API.

Given an LwIP port, any application written for LwIP can be linked without modification.

2.5 ESP32

ESP32 is a low-cost, low-power SoC with a Wi-Fi support. It was created by Espressif Systems and there are different versions for different purposes. Each board of the family could be used for this project.
The board that was used in the project (ESP-WROOM-32) has the following specs:

- Dual-core Xtensa LX6 32 bit (frequency up to 240MHz)
- 520KB of SRAM
- 4 MB of Flash memory
- Wi-Fi 802.11n
- Bluetooth 4.2
- Peripherals: SPI, I2C, I2S, UART, CAN 2.0, Ethernet MAC, ADC 12 bit
- 32 programmable GPIOs
- Hardware True Random Number Generator

Figure 2.4. also represents a functional view of a generic ESP32

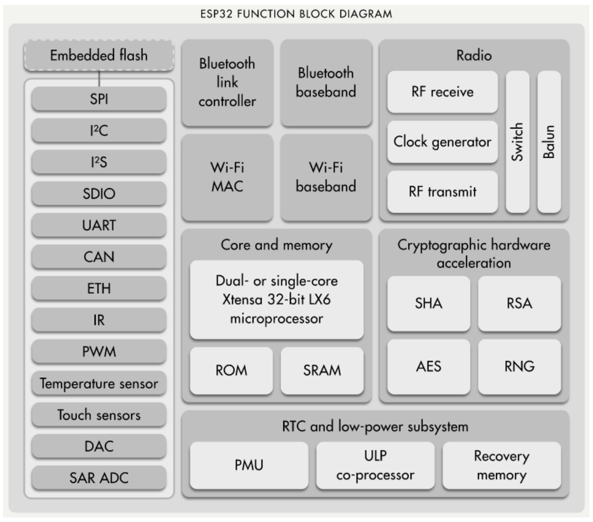


Figure 2.4: ESP32 functional block diagram.

The TRNG and the WiFi module are fundamental for the project. The first one is used in the cryptographic algorithms described in section 2.2.2 to generate a true random source and the latter to create the VPN connection based on lwIP.
ESP-IDF can be executed on this board as could be seen in the following subsection

2.5.1 ESP-IDF

ESP-IDF is the development framework used for Espressif SoCs. It contains all the APIs for the ESP32 and scripts to operate the toolchain.
This tool provides a version of FreeRTOS, which is based on Vanilla FreeRTOS v10.4.3
The other software required for development with ESP-IDF are:

- The toolchain required to compile the code for ESP32
- Build tools - CMake and Ninja to build a full application

Figure 2.5 illustrates how the application is built and uploaded on the board.

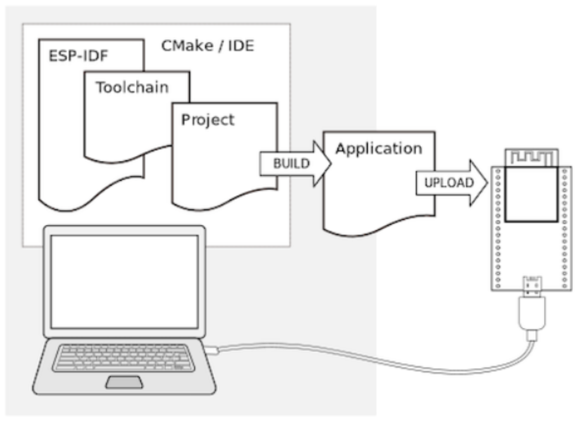


Figure 2.5: ESP-IDF software flow.

More details available on the official documentation [10].

CHAPTER 3

Implementation Overview

This chapter describes, at a high level, how the ESP32 project works, and all of the steps taken during the development phase.

3.1 Goal of the project

The focus of the project is on an implementation of a VPN client on FreeRTOS using an lwIP stack. In order to obtain it, some implementation of VPN has been analysed and, among these, as already said in previous chapter, WireGuard was chosen. Once the VPN is decided, the goal is to port it on a platform that supports FreeRTOS. The possibility considered are three:

1. Simulate FreeRTOS on Windows or Linux
2. Emulate a board behaviour on Qemu
3. Run FreeRTOS on a physical board

The starting point was the 3rd option, using the ESP32 board, since it has everything you need.
The firmware that has to be obtain therefore must interacts with a WireGuard server and must be able to send and receive data with the latter in a secure way, so guaranteeing confidentiality, integrity and authentication.
After having clarified the objectives set, in the next section it's shown how the client VPN is actually implemented.

3.2 WireGuard software module

The software module can be decided in 2 different components:

1. An internal network interface, that uses the lwIP stack and it is interfaced to receive data on a virtual internal connection, using a private IP, with the internal application
2. Internal cryptographic part, which encrypts and decrypts the data received
3. A peer part, which communicates the encrypted message through the WiFi module of the board on the external network, sending the data to the server

Therefore, the software module is placed between the application and the external server. This could be observed more in details below.

3.3 VPN client-server architecture

In order to create the VPN connection, the data must be encrypted before reaching the external network, indeed as can be observed in figure 3.1. On the ESP32 board there is an implementation of the WireGuard software module that is able to receive, through a TCP connection, data from an application that want to send a packet to the server.
The software module receive the data on his virtual network lwIP interface and encrypts the data using the algorithms described in section 2.2.2. Once the data are encrypted, they are sent through the virtual peer of the module (using an UDP connection) to the external network using the WiFi module of the board with a public IP. The data sent is encrypted and only the destination WireGuard server can decrypt them, indeed, when the latter receives the packet on his public IP, it is able to decrypt the message and send it to the internal local network with VPN.
The same mechanism works similarly if the application receives data from the server, with the WireGuard module that decrypts the data received.

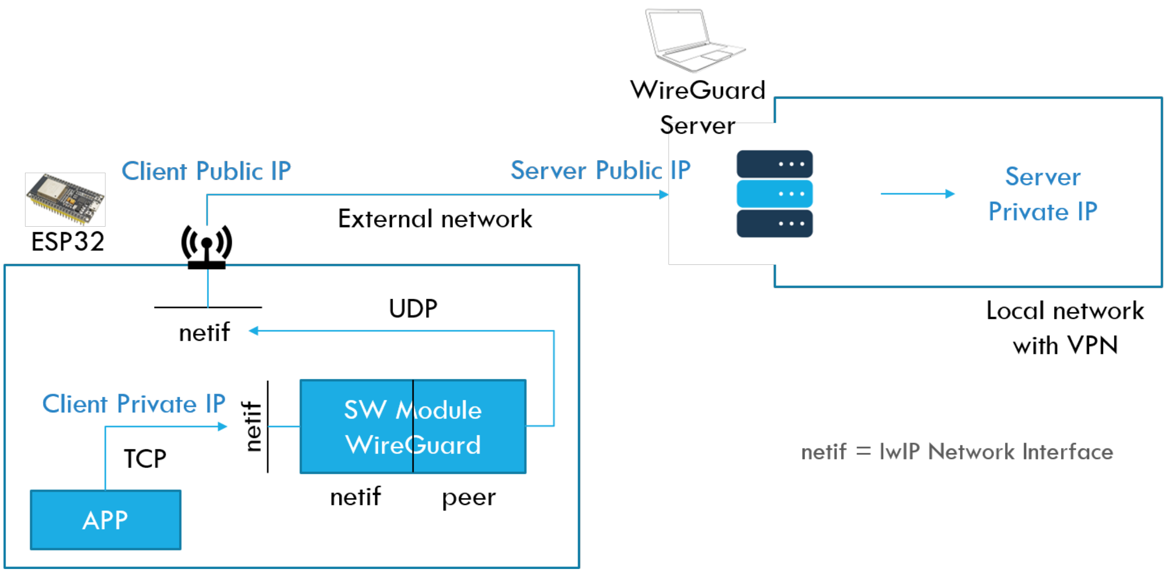


Figure 3.1: Connection between ESP32 and server through VPN.

CHAPTER 4

Implementation Details ESP32

In this chapter are reported all steps that have been done in order to understand the functionality of ESP-IDF provided by Espressif[10] in order to achieve the goal of the assignment and create the final project presented at the end of the course. After a general brainstorming, an ESP32 was decided as the chosen board because many useful resources are freely available. Mainly ESP-IDF that is Espressif's official IoT Development Framework for the ESP32, ESP32-S and ESP32-C series of SoCs. It provides a self-sufficient SDK for any generic application development on these platforms, using programming languages such as C and C++.

4.1 WireGuard on ESP32

In order to make the ESP32 board able to connect through a VPN tunnel, it is necessary to set up a WireGuard module. This module, as described before, is used to interface the board with the remote server, so related IPs needs to be consistent with the one in the server.
The WireGuard module `trombik_esp_wireguard` was taken from the Trombik project [11] and added in the directory `Esp32_tcpprompt_wg/components`.
Then the following line:

```
{REQUIRES trombik\_esp\_wireguard nvs\_flash}
```

was inserted in the file `CMakeLists.txt` in the directory `Esp32_tcpprompt_wg/main`.

4.1.1 Related projects

First of all, it is better to introduce the main projects that was the starting point reference:

- Smartalock [12], which is a C WireGuard implementation for lwIP, which cointains a generic structure (i.e. independent from the platform). As explained in its Readme document the code has four main portions:
  - `wireguard`, which contains the bulk of the WireGuard protocol code and is not specific to any particular IP stack
  - `wireguardif`, which contains the lwIP integration code and makes a netif network interface and handles periodic tasks such as keepalive/expiration timers
  - `wireguard-platform`, which contains the definition of four functions to be implemented per platform

  - `crypto`, which supports:
    * BLAKE2S
    * X25519
    * CHACHA20-POLY1305

  More details on the cryptographic algorithms are discussed in the background chapter (2.2.2).

- Trombik [11], is an alpha extension of the previous project, and creates a single WireGuard peer connection using ESP-IDF. This module is also a component of the project that has all the necessary to adapt the Smartalock to an ESP32 board.

4.1.2 WireGuard module

In order to set up the module, the starting point was the ping example present in the Trombik project [11], in order to see if the WireGuard server was reachable. To do that, has been tried the provided example configured with all the parameters like the one on the server. This example could be divided in:

- WireGuard configuration and setup, where all the WireGuard parameters are initialized in `wireguard_setup()`, setting:
  - private_key
  - listen_port
  - public_key
  - allowed_ip
  - allowed_ip_mask
  - endpoint
  - port
  - persistent_keepalive (i.e. periodically send keepalive packets)

  and then are called:
  - `esp_wireguard_init()`, which received the previous variables
  - `esp_wireguard_connect()`, connects to the corresponding peer
- WiFi and ping, that connects the module to the WiFi and manages the ping communication with server. The WiFi is connected using `wifi_init_sta()` that calls one of the two following functions:
  - `wifi_init_tcpip_adaptor()`
  - `wifi_init_netif()`

  the ping instead is managed by `start_ping()`, which communicates with the server and respond with a timeout or a success.
- time synchronization, necessary to agree during the communication with the server (it is managed by the sync_time.c)
- main, which manages the previous sections in order to run the example and ping the server.

Starting from this configuration, the example provided is used as a reference to build the application.

4.1.3 Failed approach

In a first attempt it was tested also another project from Zmeiresearch [13], but it was unfeasible to make it work. In a comparison with Trombik it was noticed that maybe it could be due to the absence of a timing synchronization.

4.2 Blinky, Wifi, Component logic

The first step was to make something run on the ESP32 to understand how the system works. Read the Espressif Documentation for ESP-IDF and for the ESP32 was essential. Following the guide provided in the Appendix A.1 , a Blinky project was tested. Its directory location is provided here:

```
esp-idf/examples/get-started/hello_world
```

By doing this the good state of the board was checked. The next step was to establish an internet connection between the ESP32 and another machine in order to add later the VPN tunnel. In order to do that a TCP/IP connection was chosen and so implemented starting from the `Hello_World` project. But before that, the WiFi connection needed to be done. The ESP-IDF provides a project where this functionality is provided.

```
esp-idf/examples/common_components/protocol_examples_common
```

In a first attempt the *protocol_examples_common* project was included as a component inside the *TCP/IP* project (to understand how to add a component into a project look at the official documentation [14])
Then the WireGuard module was analysed starting from two projects described in section 4.1 in more details. Those two projects were tested standalone and at the end the Trombik[11] project was added as a component in the TCP/IP project. As written in the next section 4.3, in the final version of the project called `Esp32_tcpprompt_wg` the WiFi functions already integrated inside Trombik project were used instead of the WiFi component provided by Espressif.

4.3 TCP/IP communication with WireGuard on ESP32

After the previous examples, that were propaedeutic to the development of the project, it was possible to implement a TCP/IP communication inside a VPN tunnel.
The project is located in the directory `/Esp32_tcpprompt_wg`.
The task developed for the communication is the following one, and it is located in the file `main.c`, inside the directory `/Esp32_tcpprompt_wg/main`:

- `static void tcp_client_task(void *pvParameters)`
  It contains the main loop of the application, in which there can be found the four main functions:

  1. `socket()` : it creates a TCP socket
  2. `connect()` : it opens the TCP connection
  3. `send()` : it sends a message
  4. `recv()` : it receives a message

  In this task it is set which port needs to be used in listening. The one used in this project is 3333.
  When the TCP socket is opened, the communication starts and makes possible to have an interaction between the esp32 and the server.
  The application that was created was a login-like communication, so the ESP32 asks for some

credentials and the user answers.
The ESP32 has a static list of available commands:

- `Hello from ESP32, please login Username:`
  It is the first message sent by the ESP32 which waits for an answers with the user credential.
- `Password:`
  After receiving the username it asks for a password, and waits for the user to answer.

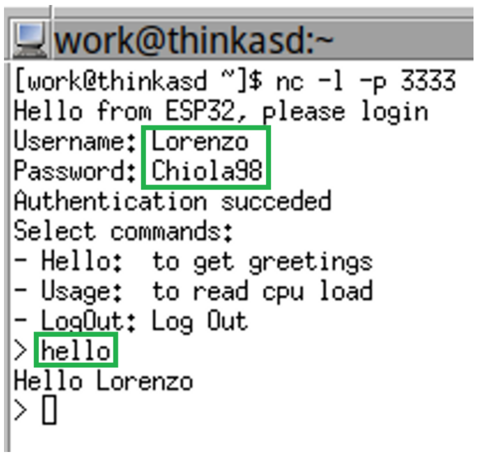
-
  ```
  Authentication succeeded
  Select commands:
  - Hello:  to get greetings
  - Usage:  to read cpu load
  - LogOut:  Log Out
  ```
  If the combination of username and password are correct, the user can choose one of these commands and will have a different answer.

- `Authentication failed, try again...  Username:`
  If the username or the password (or both) is wrong, the authentication failed and the esp32 asks again for the credentials.

After developing this task, the WiFi module and the WireGuard module were taken from Trombik[11] and added to our project.
When the project is run, the function `main()` is called and it calls the functions:

1. `wifi_init_sta()` : it initializes the WiFi module
2. `wireguard_setup()` : it setups the WireGuard server
3. `xTaskCreate()` : it is a FreeRTOS function that receive a task that needs to be created. In this case it receives the `tcp_client_task`

It also checks the time for the timing synchronization, that is necessary for the handshake.
An example of the communication can be seen in figure 4.1.



```
[work@thinkasd ~]$ nc -l -p 3333
Hello from ESP32, please login
Username: Lorenzo
Password: Chiola98
Authentication succeded
Select commands:
- Hello:  to get greetings
- Usage:  to read cpu load
- LogOut: Log Out
> hello
Hello Lorenzo
> 
```

Figure 4.1: Communication output

It could be seen that the messages sent by the esp32 are the one listed before, and for each of them there is the user answer (the message in the green box).

4.4 Adding the WireGuard module to iperf and testing performance

Between the code examples of ESP-IDF there is an iperf port for the ESP32, available at
`examples/wifi/iperf/`
(the actual iperf code is inside a submodule located at `examples/common_components/iperf/`).
iperf is a common tool used to measure network throughput between two hosts. One will run the tool in server mode, the other in client mode. Random packets of data will be exchanged by the two instances (over TCP by default) to measure the average speed.
The iperf port by Espressif is based on another one of their code examples: a command line interface provided over the serial port. Its code is available at
`examples/system/console/advanced/components/`, which is included as a submodule in iperf.
The available commands are iperf itself, `sta` and `ap` to configure the WiFi connection (station or access point), and `free` and `heap` to measure free ram (in the present or the historical low). A new command can be registered to the console, together with a callback which works much like `main(int argc, char* argv[])` with arguments passed from the command line.
The following procedure shows how to add the WireGuard module from the repository [11] to iperf for the ESP32, also adding a command to the text interface.
The result should be similar to the contents of the folder `iperf_wg/` of the supplied code. It can be useful to compare this folder to the iperf code from ESP-IDF to highlight the differences introduced with the WireGuard module.
It is assumed that the provided iperf example was built as a standard ESP32 project without issues.

1. Mkdir `components/` and git clone [11] in that folder. The `components/` folder, if it exists, is automatically searched for submodules by the build system. The only constraint is that a valid `components/*/CMakeLists.txt` file has to be present for each submodule.

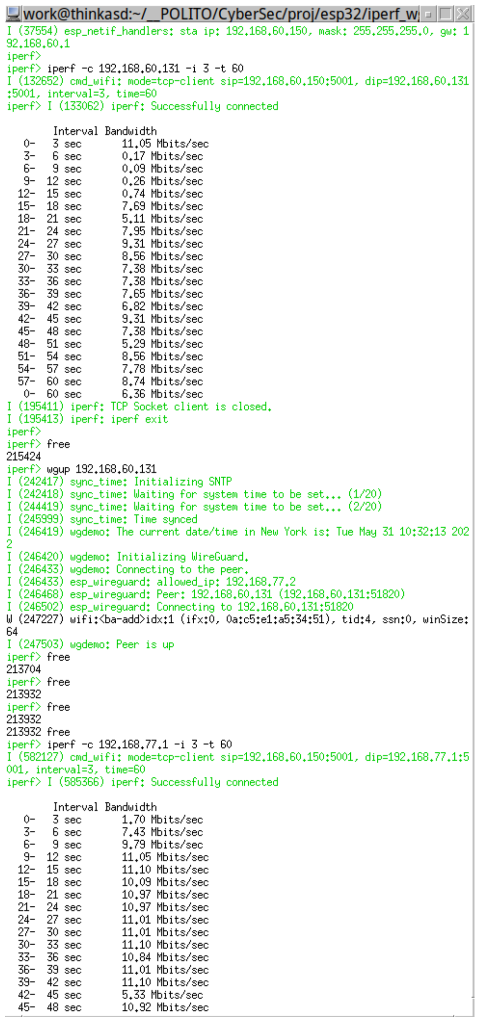
2. Move the time synchronization code
   `components/esp_wireguard/examples/demo/main/sync_time.c / .h` to `main/`. The rest of the `examples/demo/` folder can be deleted.

3. Register a new command to set up the VPN. Take `main/cmd_wifi.c / .h` as an example to adhere to the structure of the underlying iperf code. Files `main/cmd_wifi.c / .h` were written this way. The new command `wgup [peer ip address]` starts the VPN tunnel to the other peer. The call-back function is `wg_up()`, while the function `register_wg()` registers the new command in the command line at start-up.

4. Integrate the new code in the project:
   - As always when adding new source files, it is necessary to add them to the `main/CMakeLists.txt` project file so they can be compiled. Add each file name in double quotes after the last C source in the `idf_component_register(SRC "...")` macro.
   - Include the new header file inside the aggregator header `cmd_decl.h`.
   - Call `register_wg()` inside `app_main()` to register the new command.

5. Optionally use the `menuconfig` system to define options for the WireGuard module. In the supplied code this system is used, so an additional file `main/Kconfig.projbuild` was created to define some new menu entries which will appear inside the configuration along with the standard options.

It will be now possible to use the command line interface through the serial port monitor. Iperf will work as before, but it will be possible to send the `wgup` command to set up the VPN tunnel, so that any further traffic originating from the inside of the esp32 will be routed through the tunnel. 4.2 is a screenshot of the serial monitor session after running iperf, starting the VPN, and running iperf again this time inside the VPN tunnel. Note that the IP address changes for the second run; also the IP address of the first iperf server is the same as the IP address of the WireGuard peer, since they were running on the same computer. This run was used to record performance and RAM usage, see 6.2.



```
I (37554) esp_netif_handlers: sta ip: 192.168.60.150, mask: 255.255.255.0, gw: 1
92.168.60.1
iperf>
iperf> iperf -c 192.168.60.131 -i 3 -t 60
I (132652) cmd_wifi: mode=tcp-client sip=192.168.60.150:5001, dip=192.168.60.131
:5001, interval=3, time=60
iperf> I (133062) iperf: Successfully connected

        Interval Bandwidth
   0-   3 sec       11.05 Mbits/sec
   3-   6 sec       0.17 Mbits/sec
   6-   9 sec       0.09 Mbits/sec
   9-  12 sec       0.26 Mbits/sec
  12-  15 sec       0.74 Mbits/sec
  15-  18 sec       7.69 Mbits/sec
  18-  21 sec       5.11 Mbits/sec
  21-  24 sec       7.95 Mbits/sec
  24-  27 sec       9.31 Mbits/sec
  27-  30 sec       8.56 Mbits/sec
  30-  33 sec       7.38 Mbits/sec
  33-  36 sec       7.38 Mbits/sec
  36-  39 sec       7.65 Mbits/sec
  39-  42 sec       6.82 Mbits/sec
  42-  45 sec       9.31 Mbits/sec
  45-  48 sec       7.38 Mbits/sec
  48-  51 sec       5.29 Mbits/sec
  51-  54 sec       8.56 Mbits/sec
  54-  57 sec       7.78 Mbits/sec
  57-  60 sec       8.74 Mbits/sec
   0-  60 sec       6.36 Mbits/sec
I (195411) iperf: TCP Socket client is closed.
I (195413) iperf: iperf exit
iperf>
iperf> free
215424
iperf> wgup 192.168.60.131
I (242417) sync_time: Initializing SNTP
I (242418) sync_time: Waiting for system time to be set... (1/20)
I (244419) sync_time: Waiting for system time to be set... (2/20)
I (245999) sync_time: Time synced
I (246419) wgdemo: The current date/time in New York is: Tue May 31 10:32:13 202
2
I (246420) wgdemo: Initializing WireGuard.
I (246433) wgdemo: Connecting to the peer.
I (246433) esp_wireguard: allowed_ip: 192.168.77.2
I (246468) esp_wireguard: Peer: 192.168.60.131 (192.168.60.131:51820)
I (246502) esp_wireguard: Connecting to 192.168.60.131:51820
W (247227) wifi:<ba-add>idx:1 (ifx:0, 0a:c5:e1:a5:34:51), tid:4, ssn:0, winSize:
64
I (247503) wgdemo: Peer is up
iperf> free
213704
iperf> free
213932
iperf> free
213932
213932 free
iperf> iperf -c 192.168.77.1 -i 3 -t 60
I (582127) cmd_wifi: mode=tcp-client sip=192.168.60.150:5001, dip=192.168.77.1:5
001, interval=3, time=60
iperf> I (585366) iperf: Successfully connected

        Interval Bandwidth
   0-   3 sec       1.70 Mbits/sec
   3-   6 sec       7.43 Mbits/sec
   6-   9 sec       9.79 Mbits/sec
   9-  12 sec       11.05 Mbits/sec
  12-  15 sec       11.10 Mbits/sec
  15-  18 sec       10.09 Mbits/sec
  18-  21 sec       10.97 Mbits/sec
  21-  24 sec       10.97 Mbits/sec
  24-  27 sec       11.01 Mbits/sec
  27-  30 sec       11.01 Mbits/sec
  30-  33 sec       11.10 Mbits/sec
  33-  36 sec       10.84 Mbits/sec
  36-  39 sec       11.01 Mbits/sec
  39-  42 sec       11.10 Mbits/sec
  42-  45 sec       5.33 Mbits/sec
  45-  48 sec       10.92 Mbits/sec
```

Figure 4.2: iperf running on the ESP32 both outside and inside the VPN tunnel. The remote computer is reachable both from inside and outside the VPN, it is running iperf in server mode: `iperf -s`

CHAPTER 5

Running the VPN in the FreeRTOS simulator for Linux

FreeRTOS already has a port that runs as a native process in Posix systems, known as the Posix simulator. LwIP also has support for running under Linux using virtual network interfaces. In this chapter LwIP is configured to run in FreeRTOS (which is a very common use case with a port already existing) while using a Linux virtual network interface. Finally, the WireGuard module is added.

5.1 Build FreeRTOS for Linux (simulator)

FreeRTOS supports running as a process inside a Posix system for simulation purposes.
The Posix port implements tasks as pthreads, however only one thread will be running at any time, dictated by the FreeRTOS queue.
Each thread receives Posix signals to simulate interrupts (mainly SIGALARM for the System Tick interrupt), so that execution is periodically passed back to the task queue, which then restarts one of the threads with SIGUSR1. See also (simulator doc), or the comments in the port files in `FreeRTOS/Source/portable/ThirdParty/GCC/Posix/` of the FreeRTOS distribution. A simulation port of FreeRTOS for Windows is also available. It was not tested for this project because the limited available networking options meant adding LwIP would have been impractical.
Note that FreeRTOS loses any real time guarantee when ported to Posix, since the latter is not deterministic. Scheduling still follows the same criteria that would apply on an embedded system, and operation with a 1 ms system tick is possible on an unloaded laptop CPU. Problems might arise inside virtual machines or on old hardware. Missing some system ticks would not be critical for this project.

5.2 Adding LwIP on top of FreeRTOS

LwIP will not be running on an embedded system, but rather in a simulated environment. The compiler will be a standard gcc for Linux, while the underlying operating system will be FreeRTOS. A Unix TAP virtual network interface will be created to allow LwIP to connect to something outside the FreeRTOS simulator.
This procedure shows how to make LwIP available to FreeRTOS applications such as the blinky example (see A.4.1 for details and build instructions). The expected result of these steps should be similar to the contents of the folder `Posix_GCC_lwip1/` of the supplied code, which was derived from blinky. In that code A TCP connection is established to another device on the network; the same messages that are displayed on `stdout` by blinky are sent over TCP too.

5.2.1 tapif: the network interface

A Unix TAP is a virtual network interface that behaves mostly like an Ethernet interface. On the OS end it behaves like any other network interface of the system, it can be configured with the `ip link / ip addr / ip route` utilities and can have IP addresses. The other end of the TAP is not a driver for a physical interface, rather it can be controlled through the `open() / close() / read() / write()` system calls by a user application that wishes to emulate a NIC.
TAPs are commonly used by VPN user space clients (not WireGuard) to provide a network interface that appears connected to the VPN local network, and by virtual machines to give network access to guest systems.
To set up a TAP in Linux operating systems see Section A.4.

5.2.2 LwIP setup

1. Download the two LwIP packages: both
   - the sources of LwIP itself [15], and
   - the contributed code [16], which contains some ports that are going to be used.
2. Write the `lwipopts.h` file:
   - thread defines: these should already be defined when starting from a `lwipopts.h` file with `NO_SYS==0`. For a ready-made example see `Posix_GCC_lwip1/lwip/lwipopts.h`. Determining the values needed without knowing the internals of LwIP essentially requires to study the internals, either from the (excellent) documentation or by debugging the program hunting the reasons for trace dumps, which are rarely obvious. If at all possible, start from an `lwipopts.h` from a similar application.
     All available parameters can be read from the defaults file `lwip/src/include/lwip/opt.h`.
   - Please also add the following lines to `lwipopts.h`:
     `#define MEMP_NUM_SYS_TIMEOUT 6 // because the default is not correct (see opt.h line 508)`
     `#define PBUF_POOL_BUFSIZE 256 // if you wish to use the WireGuard module, because of an assumption done in that code, see` 5.3
3. Add the sources of LwIP to the project
   - mkdir `lwip`
   - copy the `src/` directory from the LwIP distribution to `lwip/`
4. Copy the port files
   - mkdir `lwip/port/`
   - copy the contents of `ports/freertos` from the LwIP contrib distribution to `lwip/port/`. These are the `sys_arch.c/h` files that implement threads, semaphores, mailboxes and mutexes based on FreeRTOS.
   - copy `ports/unix/port/include/arch/cc.h` from the LwIP contrib distribution to `lwip/port/include/`. This header contains the basic definitions for compatibilitiy with the Linux-native `gcc` compiler.
   - copy `ports/unix/port/netif/` from the LwIP contrib to `lwip/port/`. Only `tapif.c` and `include/tapif.h` are really necessary.

- remove error handling: comment the call to `perror()` at `line 427 in tapif.c`.
  This netif was built for LwIP over straight Linux, but we are running LwIP over FreeRTOS inside Linux. The system tick in the FreeRTOS port for Linux is implemented using a Posix Signal that is generated every millisecond to interrupt the running task and run the scheduler. When the signal is delivered it interrupts any blocking system call, in particular the system call used by tapif to communicate with the Unix TAP returns `EINTR`. The call can be restarted without special considerations, so the error can be safely ignored.

5. Add the necessary LwIP files to the variables `SOURCE_FILES` and `INCLUDE_DIRS` in the `Makefile`. The list was determined by trial and error, building the project and resolving symbols that were missing while linking, and by referencing some examples. Note the use of wildcards in the `Makefile`:
   - `lwip/src/core/*.c`, `lwip/src/core/ipv4/*.c`: basic LwIP functionality.
   - `lwip/src/api/*.c`: this project uses the BSD socket API, which is the most abstract, so all the files here are needed. A project which only uses the `RAW` or `netconn` APIs could shed some size by not building the socket API.
   - `lwip/port/sys_arch.c`: the FreeRTOS port layer.
   - `lwip/src/netif/ethernet.c` which is almost always used by ethernet-based netifs, and `lwip/port/netif/tapif.c` which is the outside-facing netif we are going to use.
   - Include directories: `lwip/src/include` for LwIP and `lwip/port/include` for the port layer (here are `sys_arch.h`, `cc.h` and optionally `perf.h`). Also include whatever directory contains `lwipopts.h` (it can be found in folder `lwip/` in the supplied code, but it could be saved in `lwip/port/include` for simplicity).

6. Write some basic application code. A finished example derived from the `Posix_GCC` FreeRTOS example is given in the `Posix_GCC_lwip1/` folder of the supplied code.
   This example does the following:
   - in `main_blinky()`:
     (a) initialize LwIP with `tcpip_init(NULL, NULL);`
     (b) create the outside-facing network interface of the tapif type with `netif_add()`
     (c) create a TCP socket (internally to LwIP) with `socket()`
   - in a separate task function called `prvTCPConnectorTask`(), run after starting the scheduler:
     (a) establish the TCP connection with an external host on port 3333 with `connect()`
     (b) in general, run any operation that communicates through a netif *only after starting the scheduler*, or else the netif will not be functional, making the operation hang or fail.

   Use Netcat to listen on TCP port 3333 on the computer which runs the simulator. Use the command `nc -v -l -p 3333`.

5.3 Adding the WireGuard module to an LwIP project

This section will guide the reader through the process of adding the WireGuard module to the basic FreeRTOS + LwIP setup obtained in the previous section.
The end result of this procedure should be similar to the contents of the folder `Posix_GCC_lwip2_wg/` of the supplied code.

1. Download the wireguard-lwip module either from [12] (original, preferred), [11] (ESP32 port, identical clone of (1)), and copy it to a folder in the project (see `wireguard-lwip` in the supplied project).

2. Ensure that `PBUF_POOL_BUFSIZE` is defined to be more than 134 in `lwipopts.h`. This is needed because the WireGuard module expects to get the handshake response (92 bytes), received from the remote peer, inside a single pbuf. The pbuf pool contains space for a number of pbufs, the data buffers of LwIP. When a packet is too big for one pbuf, a singly linked list of pbufs is created to hold the whole message. This quantity of data also has 42 bytes of overhead for Ethernet, IP and UDP, totalling to 134 bytes.

3. Create a platform support file (`wireguard-lwip/src/wireguard-platform_unix_freertos.c` in the project code). Implement in this file four functions needed by the WireGuard module. Note that you can just use the example `wireguard-lwip/src/example/wireguard_platform.c` as a basic, workable but insecure version. The four functions are:

   - `wireguard_random_bytes()` should be a strong random number generator. Note that **the default example is a weak pseudo random number generator, unsuitable for real-world usage**. On a Linux platform it could be implemented to read /dev/random, however this involves system calls which are easily broken by the FreeRTOS simulator, as was shown for the tapif interface in the last chapter. In this case the `read()` call could be interrupted, making the function return uninitialized values or some predefined value, like zero, which would critically weaken the cryptographic system anyway, unless a more complete solution is developed.
   - `wireguard_sys_now()` just returns LwIP's `sys_now()` which is a 32 bit unsigned number of milliseconds elapsed from an epoch (in this port the epoch is the start of simulation). Note that this time will wrap around after less than 50 days.
   - `wireguard_tai64n_now()` returns the tai64n timestamp (specification: [17]) used in the WireGuard packets. It is essential that this timer increases for every new packet, otherwise the remote peer will just drop the packets.
     The example platform file implements this function using `sys_now()`, which would break the connection after 49 days, until the server is killed and restarted.
     `wireguard_tai64n_now()` was edited to make it return the current time of the underlying Linux OS. The solution uses the `time()` C library function which, in principle, could generate system calls. However, most of the time no system calls are made, and the function is only used on handshakes, which happen every two minutes by default. For these reasons the probability of an error breaking a `time()` call is low, and the solution was demonstrated to work well in practice. Using the correct time also avoids the annoyance of restarting the server every time the simulation is killed and restarted, since the timestamp keeps increasing instead of resetting to 01/01/1970.

   - `wireguard_is_under_load()` just returns false. This could be made to return true if the system is experiencing load above some threshold. When true, the WireGuard module will send out cookies instead of handshake responses, and only accept handshakes having a non-zero mac2 field, which needs the cookie to be computed (see the DoS mitigation section of the protocol description [18]). This mechanism would provide limited protection against overload and DoS attacks, since handshaking is relatively expensive compared to ordinary communication.

4. Initialize the tunnel in the application code, the same way as it is done for the ESP32 platform (see `wireguard-lwip/src/esp-wireguard.c` from the repository (2) which was also used for

the ESP32.).
The code cannot be copied exactly from the ESP32 port however, because it uses many functions from `esp-idf`.
The same operations were adapted into the task `prvTCPConnectorTask()` and the function `unix_wireguard_setup()`. The operations performed are:

- netif_add() to create an internal netif of the type used by `wireguard-lwip`. This will be the VPN termination inside the the target device (simulator in this case).
- `wireguardif_add_peer()` to configure the connection to the remote peer
- `wireguardif_connect()` to initiate a handshake with the remote peer. This phase can be delayed until the connection is actually needed, as is the standard behavior on the Linux and Windows implementations.
- wait for the tunnel to be open by calling `wireguardif_peer_is_up()`

It may be useful to also resolve the IP address of the remote peer as in

`wireguard-lwip/src/esp-wireguard.c`.

5.4 Failed approach

The initial goal of the project was to have a working VPN inside FreeRTOS, so the first networking stack to be considered was FreeRTOS's own FreeRTOS+TCP (the name is somewhat misleading, since it is a complete networking stack). It was not possible for the team to compile a usable demo for Windows or Linux. In particular the virtual network interface based on PCAP was thought to be the culprit.
PCAP is the library used by WireShark [19] to capture traffic from a network interface on many operating systems. It is used to provide access to the network in the FreeRTOS+TCP Linux and Windows simulators, by reading and injecting packets from and to a physical Ethernet interface of the host system.
A PCAP netif also exists for LwIP on Linux (`ports/unix/port/netif/pcapif.c` in the contrib package), but it was discarded in favour of the tapif because of the experiences with PCAP in FreeRTOS+TCP. It was also not clear if PCAP could attach to a virtual TAP interface for experimenting inside a single computer or virtual machine, while the tapif is built for that.

CHAPTER 6

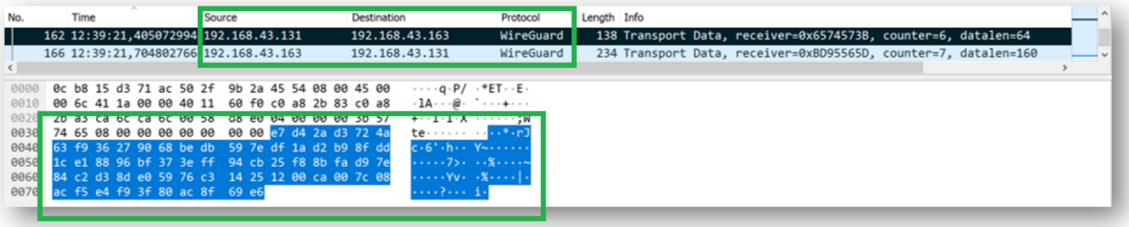
Results

This chapter demonstrates that the communication between the Client and Server is encrypted as expected, as well as the obtained performance on the ESP32 board. All issues identified during the project's development are reported and explained. A small section with possible future works is also described.

6.1 Encrypted communication checking

After developing the project application, the packets sent during the communication where analysed through WireShark [19], which is a free and open source packets analyser. In this way it was possible to test that the interaction between the ESP32 and the user was really under a VPN protocol, and so correctly encrypted.
Below there are two WireShark capture: in these the packet that was captured was the one containing the user password, because it is the most vulnerable data exchanged.

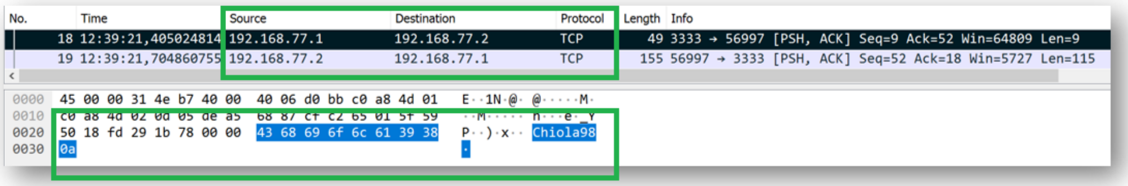
- In Figure 6.1 there is the capture of the packet in the public network.
  It could be seen that the source and destination IPs are the one of the physical devices, and the communication is happening under a WireGuard protocol.
  Moreover the content of the packets is not visible, and so it has been successfuly encrypted.
- In Figure 6.2 there is the capture of the packet in the WireGuard Network Interface.
  In this case the source and destination IPs are the one assigned by WireGuard Interface, and it could be seen that the packet sent is exactly the user password, so the packet exchange during the communication is the correct one.



No.	Time	Source	Destination	Protocol	Length	Info
162	12:39:21,405072994	192.168.43.131	192.168.43.163	WireGuard	138	Transport Data, receiver=0x6574573B, counter=6, datalen=64
166	12:39:21,704802766	192.168.43.163	192.168.43.131	WireGuard	234	Transport Data, receiver=0xBD95565D, counter=7, datalen=160

```
0000  0c b8 15 d3 71 ac 50 2f  9b 2a 45 54 08 00 45 00   ····q·P/ ·*ET··E·
0010  00 6c 41 1a 00 00 40 11  60 f0 c0 a8 2b 83 c0 a8   ·lA···@· `···+···
0020  2b a3 ca 6c ca 6c 00 58  d8 e0 04 00 00 00 3b 57   +··l·l·X ······;W
0030  74 65 08 00 00 00 00 00  00 00 e7 d4 2a d3 72 4a   te······ ····*·rJ
0040  63 f9 36 27 90 68 be db  59 7e df 1a d2 b9 8f dd   c·6'·h·· Y~······
0050  1c e1 88 96 bf 37 3e ff  94 cb 25 f8 8b fa d9 7e   ·····7>· ··%····~
0060  84 c2 d3 8d e0 59 76 c3  14 25 12 00 ca 00 7c 08   ·····Yv· ·%····|·
0070  ac f5 e4 f9 3f 80 ac 8f  69 e6                     ····?··· i·
```

Figure 6.1: WireShark capture public Network



```
No.  Time                  Source          Destination     Protocol  Length  Info
  18 12:39:21,405024814    192.168.77.1    192.168.77.2    TCP           49  3333 → 56997 [PSH, ACK] Seq=9 Ack=52 Win=64809 Len=9
  19 12:39:21,704860755    192.168.77.2    192.168.77.1    TCP          155  56997 → 3333 [PSH, ACK] Seq=52 Ack=18 Win=5727 Len=115

0000  45 00 00 31 4e b7 40 00  40 06 d0 bb c0 a8 4d 01   E··1N·@· @·····M·
0010  c0 a8 4d 02 0d 05 de a5  68 87 cf c2 65 01 5f 59   ··M····· h···e·_Y
0020  50 18 fd 29 1b 78 00 00  43 68 69 6f 6c 61 39 38   P··)·x·· Chiola98
0030  0a                                                 ·
```

Figure 6.2: WireShark capture WireGuard Network Interface

6.2 Performance and resource consumption

No performance measurements were made on the FreeRTOS simulator for Linux, since it is very different from an actual embedded device. Program sizes are also influenced by C library code needed to interface with the host, since the whole simulation is compiled into a single executable file.
Performance was evaluated on the ESP32 platform using the `iperf` port provided in the `ESP-IDF` SDK with the `wireguard-lwip` module added, see chapter 4.4.
The WiFi network was too slow in the experiments conducted to determine if the cryptography was CPU bound. The best throughput achieved was 11 Mbit/s both directions, both inside and outside the WireGuard tunnel.
While this result is not conclusive, it clears the envelope for many applications with limited bandwidth requirements. Also, many real-world scenarios replicate the same conditions yielding the same or much worse link speeds. All in all, the communication will not be CPU-bound in most practical cases.
Resource consumption was low enough for the ESP32, although this should be evaluated for each platform.
37 kilobytes of ROM are needed for the WireGuard module, including the very basic configuration command described in the dedicated chapter. This was deduced from binary sizes at compile time: `idf.py build` reports a number of used bytes in hexadecimal after the build is complete.
For an ESP32 this should be an acceptable quantity of flash for most projects, especially considering the operating system requires 600 kB of flash.

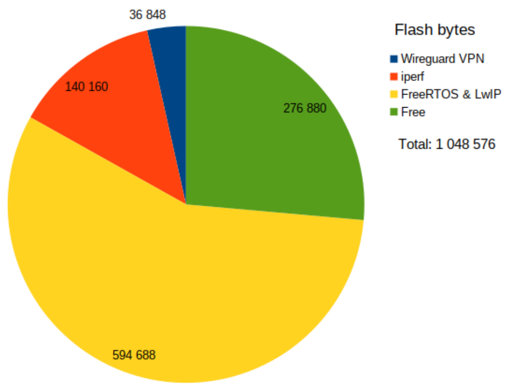


Figure 6.3: iperf with WireGuard: flash memory footprint contributions

RAM heap usage on the ESP32 was 1700 bytes at idle and 7 kilobytes while running iperf. This was reported using the `free` and `heap` commands available in the serial console of the iperf port, see

4.4 and 4.2.
The ESP32 has 500 kB of internal RAM, of which 220 kB are usable by applications.

6.3 Known Issues

This report shows how to set up a basic VPN tunnel using WireGuard on different cases, however some issues were left unfixed.

- Every procedure described works as intended on Linux systems only. On Windows platforms WSL is available, however it does not provide a complete Linux system, and a common virtual machine could be more useful. Most notably:
  - WSL 2 is basically a virtual machine, so networking is more complicated (a virtual interface connects to the internet through NAT, **hyper-v virtual ethernet adapter** issues) and no server can easily run from inside the VM. Also connecting serial devices (which is needed to flash a firmware onto ESP32 chips) requires setting up USB pass-through every time the device is plugged in, which sums up to one command in the Windows host and one command inside WSL.
  - WSL 1 offers better compatibility. There were no problems with networking, but USB devices problem still remain. It is hard to fix and even so, the build process was really slow (10-15 min on a performant machine).
  - Oracle VirtualBox was most practical, offering one-click USB pass-through configuration, and bridged networking that was functional over WiFi too.

  Networking was especially important, because some tools (most notably Netcat) are missing in Windows.

- 32 bit `sys_now()` timestamp: on most LwIP platforms the TAI64N timestamp necessary for WireGuard is obtained from the `sys_now()` function of LwIP that returns a 32 bit unsigned millisecond count that overflows every 49 days. The timestamp is always between 01/01/1970 and 18/02/1970. While it is not a problem that the date is far in the past, when the timestamp overflows, or when the device reboots, the date and time go back to a previous instant (01/01/1970). The WireGuard server will reject all handshake packets with an earlier timestamp than the last received, so communication will not be possible until the server is killed and restarted.

- The random number generator used in the Linux simulator port is not cryptographically secure. See the `rand()` call inside `wireguard_random_bytes()` in file

  ```
  wireguard-lwip/src/wireguard-platform_unix_freertos.c
  ```

  Although security was not needed for the simulation, it should be clear that this flaw compromises the whole cryptosystem. A strong random number generator should be implemented for any practical application, as was done for the port to the ESP32 platform.

- A large number of TCP retransmissions is sometimes generated by the peer using the **wireguard-lwip** module, usually for an interval of up to 10 seconds, even in perfect conditions (like TAP networking). While performance is very good, this issue should probably be addressed before a practical application is deployed, as there is the potential for some important flaw to be the cause of such a seemingly random malfunction.

- All the test on the esp32 have been done using the ESP-IDF version 4.4.1. The latest version available at the time of writing this paper (v5.0.0) does not work with the presented project.

6.4 Future Works

The following two important steps are still missing to produce a usable solution:

- User friendly configuration of the VPN keys and static IPs. This can be added along the configuration of the WiFi access parameters, keeping in mind that it may not be practical to type two 44-character long keys on a small keyboard or on a number-only keyboard.
- Internet routing was not tested nor desired for this project, communication with a single computer was enough. While potentially no changes to the `wireguard-lwip` module may be needed, the configuration becomes more involved on any peer that has to route traffic to other hosts inside the VPN, be it on a physical local network or to other connected WireGuard peers.

CHAPTER 7

Conclusions

Starting with the ESP32 board and then porting in simulation on Linux FreeRTOS, the goal of implementing a VPN Client for FreeRTOS was met. Several issues arose during the project's development. The first step was to research what a Virtual Private Network is, as well as its functions and how to use it.
Following some evaluations, WireGuard was selected as the protocol to create an encrypted communication that ensures confidentiality, integrity, and authentication. The ESP32 was chosen as the client's development board because it provides all of the required utilities at a low cost. The Espressif's official IoT Development Framework (ESP-IDF) functionalities were studied in order to use them on the board.
While browsing the internet, examples of previously created WireGuard implementations on IoT devices were discovered and used as a starting point for the project. To simulate a simple packet transmission, a TCP/IP port on a Linux machine was opened, and the ESP32 was used as the sender. Following the successful communication, the WireGuard module was added to ensure a secure channel. To ensure that the packets were encrypted, an Internet traffic analyser was used. So, after the previously described implementation, it was easier to understand how the WireGuard module works and how to configure it. This enabled it to be ported in a more general scenario using the FreeRTOS simulator for Linux.
All of the work done allowed the establishment of a raw TCP/IP communication, but a more complex application exploiting the full potential of a VPN could be developed. Other improvements must be made before this solution can be used in a real-world scenario.

APPENDIX A

User Manual

A.1 ESP32 System Setup

In this Section, the set of both hardware and software resources required to set up the ESP32 are outlined. At the end of this Section, you will have acquired a clear overview of the prerequisites to set up the environment and run a project. It is strongly recommended to use a virtual machine (use Virtual Box[20]) or machine with a Linux distribution (i.e. Ubuntu Linux[21]) installed as operating system and to follow this guide to set up everything.

A.1.1 Hardware resources

- **ESP32 board**.
- **USB cable** - USB A /micro USB B
- **Computer** running Windows, Linux, or macOS

This document will focus only on the implementation for the Linux environment. In case of using a virtual machine, make sure that the USB ports are visible to your system and internet connection is provided by WiFi bridge (take a look in Known Issues 6.3).

A.1.2 Software resources

You need the following tools:

- **Toolchain** to compile code for ESP32
- **Build tools** - Cmake and Ninja to build a full Application for ESP32
- **ESP-IDF** that essentially contains API (software libraries and source code) for ESP32 and scripts to operate the Toolchain
- **IDE** for C language, on your choice, to edit better the code (OPTIONAL, not covered in this report)

A.2 Guide for Linux

This section provides a summary on getting everything installed and ready to use on Linux (Ubuntu or Debian). All the reported material is based on the official guide provided by Espressif[22] end various trials.

NOTE: Install process of Oracle VirtualBox VM and a Linux distribution on it, is not reported in this paper.

A.2.1 Step 1 - Install Prerequisites

In order to use ESP-IDF with the ESP32, some software packages need to be installed. Open a terminal and run the following command:

```
sudo apt-get install git wget flex bison gperf python3 python3-venv pip
cmake ninja-build ccache libffi-dev libssl-dev dfu-util libusb-1.0-0
```

A.2.2 Step 2 - Get ESP-IDF

To build applications for the ESP32, software libraries provided by Espressif are needed.
Download it from here [23, ESP-IDF]
Or open a Terminal, and run the following commands:

```
mkdir -p ~/esp
cd ~/esp
git clone -b v4.4.1 --recursive
    https://github.com/espressif/esp-idf.git esp-idf-v4.4.1
```

NOTE: Is strongly recommended to use the ESP-IDF v4.4.1

A.2.3 Step 3 - Set up the tools

Aside from the ESP-IDF, you also need to install the tools used by ESP-IDF, such as the compiler, debugger, Python packages, etc, for projects supporting ESP32.

```
cd ~/esp/esp-idf-v4.4.1
./install.sh ESP32
```

A.2.4 Step 4 - Set up the environment variables

This step **must** be executed every time a new Terminal is opened in order to build, flash, etc.
In the terminal where ESP-IDF is going to be used, run:

```
. $HOME/esp/esp-idf-v4.4.1/export.sh
```

A.3 Start the TCP/IP WG project on ESP32

This project is located in the **ESP32_tcpprompt_wg** directory.

Before continue in this section is recommended to set up the Wireguard server, a small guide is provided in section A.5 of the Appendix. Is strongly recommended to use a Linux Machine or an Android device as a Server. Remember to generate the Keys, see in section A.5 This is because for the simple TCP/IP communication built for this project, Netcat is needed to listen on a specific port and it is only available for Linux and Android. Here are reported the Crypto key Routing Tables WireGuard network interface on both client and server side.

Server:

```
[Interface]
Address = 192.168.77.1/24  # Network Interface IP addr. of the Server
ListenPort = 51820  # default Listening port for UDP communication
PrivateKey = GIqk1aeXqvV1WK6P8+1tplf7JDSUvl46sh02cVHPY2A=  # sK Server

[Peer]
PublicKey = avs2h9rnmboc0o95q/zFEVOa3fFxVNqhW5kE83Zksm4=  # pK Client
AllowedIPs = 192.168.77.2/32  # Network Interface IP addr. of the Client
```

Client:

```
[Interface]
Address = 192.168.77.2  # Network Interface IP addr. of the Client
ListenPort = 51820  # default Listening port for UDP communication
PrivateKey = qEnqWM2tORDVhvTZeGlK+c5jGxiFeVBRLiFPx8iAtXg=  # sK Client

[Peer]
PublicKey = dk0lTzfwjmWHtggRkw8rmUkQQqox26G8QSBAdXebAwQ=  # pK Server
AllowedIPs = 0.0.0.0  # So that all traffic goes through the tunnel
```

The IP Addresses of the Network interfaces 192.168.77.1 for the Server and 192.168.77.2 for the client were chosen a priori, any other available IP Address can be used. The Private and Public keys of each Peer were generated using the code provided in this section A.5.

NOTE: On the client side, so on the ESP32 in this case, also the **Endpoint** must be configured. During the tests the router assigned to the ESP32 the IP Address:'192.168.43.131'. In this case Endpoint: '**192.168.43.131:51820**'

A.3.1 Configure the Project

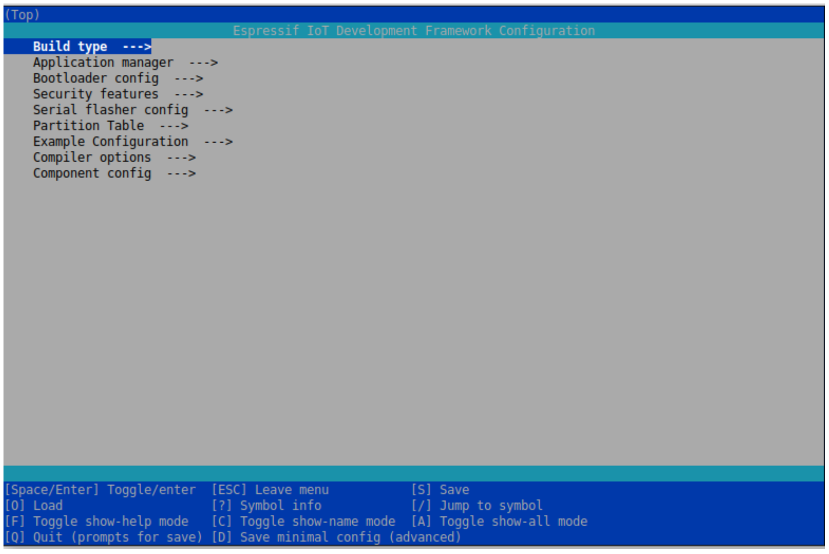
Remember to set the environment variable A.2.4 if not done. Copy the project `/ESP32_tcpprompt_wg` to `~/esp` directory:

```
cd ~/esp
cp -r $IDF_PATH/ESP32_tcpprompt_wg .
```

Here there should be `esp-idf` and `ESP32_tcpprompt_wg` folders in the same directory. Navigate to the project directory, set ESP32 as the target (To do only the first time), and run the project configuration utility menuconfig in order to set up your configuration.

```
cd ~/esp/ESP32_tcpprompt_wg
idf.py set-target ESP32
idf.py menuconfig
```

After the `idf.py menuconfig` command, the following window should appear A.1. In this window many basic parameters of the ESP32 can be set.



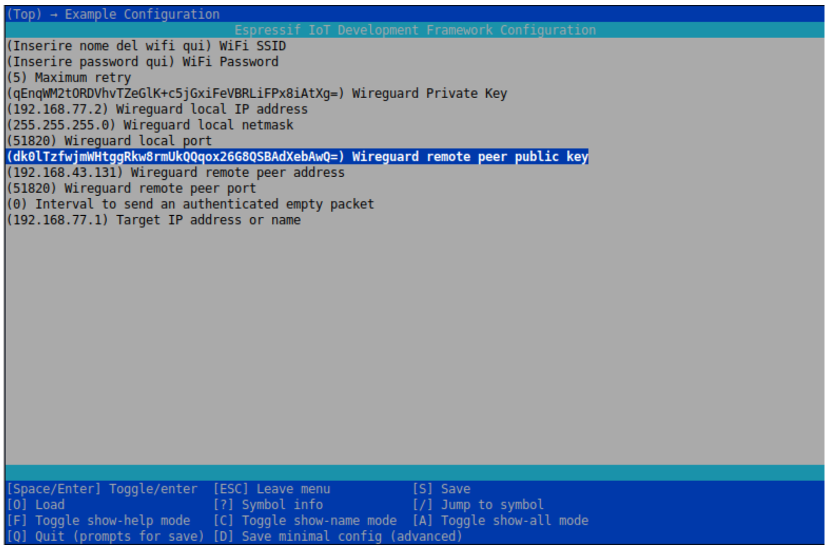
```
(Top)
                         Espressif IoT Development Framework Configuration
    Build type  --->
    Application manager  --->
    Bootloader config  --->
    Security features  --->
    Serial flasher config  --->
    Partition Table  --->
    Example Configuration  --->
    Compiler options  --->
    Component config  --->

[Space/Enter] Toggle/enter  [ESC] Leave menu           [S] Save
[O] Load                    [?] Symbol info            [/] Jump to symbol
[F] Toggle show-help mode   [C] Toggle show-name mode  [A] Toggle show-all mode
[Q] Quit (prompts for save) [D] Save minimal config (advanced)
```

Figure A.1: Menuconfig window of the project

Move through the menu using arrow keys end go in **Example Configuration** to set your personal settings (Wifi connection) and the peer configuration of WireGuard module (or leave it as it is, to use our configuration) A.2.

Remember:

- **Wireguard Private Key**: set the private key of the client peer
- **Wireguard remote peer public key**: set the public key of the peer of the server
- **Wireguard local IP address**: set the static IP address of the client peer
- **Wireguard remote peer address**: set the IP address assigned by the router to the server (see NOTE in section A.3)
- **Wireguard local/remote peer port**: set the ports for the UDP comunication
- **Target IP address or name**: set the static IP address assigned to the peer server from the server configuration A.5



```
(Top) → Example Configuration
                    Espressif IoT Development Framework Configuration
(Inserire nome del wifi qui) WiFi SSID
(Inserire password qui) WiFi Password
(5) Maximum retry
(qEnqWM2tORDVhvTZeGlK+c5jGxiFeVBRLiFPx8iAtXg=) Wireguard Private Key
(192.168.77.2) Wireguard local IP address
(255.255.255.0) Wireguard local netmask
(51820) Wireguard local port
(dk0lTzfwjmWHtggRkw8rmUkQQqox26G8QSBAdXebAwQ=) Wireguard remote peer public key
(192.168.43.131) Wireguard remote peer address
(51820) Wireguard remote peer port
(0) Interval to send an authenticated empty packet
(192.168.77.1) Target IP address or name

[Space/Enter] Toggle/enter  [ESC] Leave menu           [S] Save
[O] Load                    [?] Symbol info            [/] Jump to symbol
[F] Toggle show-help mode   [C] Toggle show-name mode  [A] Toggle show-all mode
[Q] Quit (prompts for save) [D] Save minimal config (advanced)
```

Figure A.2: Example configuration window in Menuconfig

A.3.2 Build the Project

Remember to set the environment variable A.2.4 if not done. Build the project by running:

```
idf.py build
```

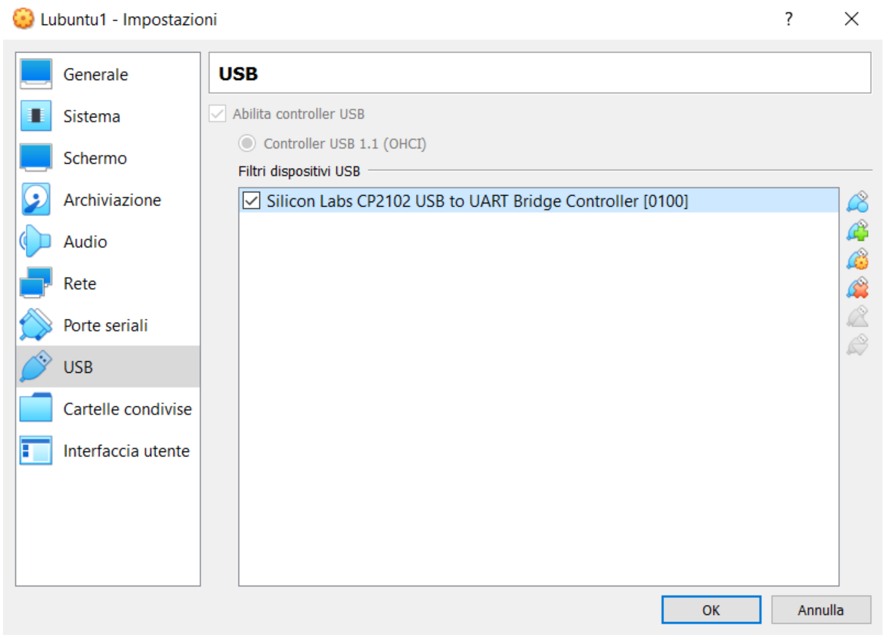
This command will compile the application and all ESP-IDF components, then it will generate the bootloader, partition table, and application binaries. The first time the command is executed will be slow, then each time a small change is done the build is going to be faster.

A.3.3 Connect the Device

Now connect the ESP32 board to the computer and check under which serial port the board is visible. On Linux the serial ports have `/dev/tty` naming patterns. So, in order to understand the name of the port assigned to the ESP32 run the command `ls /dev/tty*` two times, before and after connecting the device and check the difference. Usually if there are no other USB connections the assigned port should be `/dev/ttyUSB0`. Add the User to the dialout group using the following command:

```
adduser <user_name> dialout
```

NOTE: In case a VM on Windows is used, remember to install the *Silicon Labs CP2102 USB to UART Bridge Controller* driver on Windows. After that, go under **Machine** ↣ **Settings** ↣ **USB** in the VirtualBox Menu bar and add a new USB filter (the plus icon) selecting the correct one


Lubuntu1 - Impostazioni

USB

Abilita controller USB

Controller USB 1.1 (OHCI)

Filtri dispositivi USB

Silicon Labs CP2102 USB to UART Bridge Controller [0100]

Figure A.3: Set the USB connection through Oracle VirtualBox VM.

A.3.4 Flash and Monitor the Device

Remember to set the environment variable A.2.4 if not done. Flash the binaries that were built onto the ESP32 board by running:

```
idf.py -p PORT [-b BAUD] flash
```

Replace PORT with the ESP32 board's serial port name (i.e. /dev/ttyUSB0).
To check if the project is indeed running, type the following command that launches the IDF Monitor application:

```
idf.py -p PORT monitor
```

Remember: To exit IDF monitor use the shortcut `Ctrl+]`.
The two commands can be combined in one:

```
idf.py -p PORT flash monitor
```

A.3.5 Run the project

Turn on the Virtual Network Interface on the Server A.5, and listen on a port using Netcat[24] using the following command on the Linux machine running the Server:

```
nc [-v] -l -p 3333
```

Or Install "nc for Android" app from the PlayStore [25] and use the above command.
NOTE: In case Netcat is note installed on the selected Linux distribution, install with the following command:

```
sudo apt-get install -y netcat
```

After everything is up, turn on the ESP32 (i.e. connecting it to the PC) with the project already flashed. Other details about the project are explained in 4.1

A.4 Start the TCP/IP WG project on FreeRTOS simulator for Linux

This project is located in the **Posix_GCC_lwip2_wg** directory. Here are reported the steps to run the project described in section 5.3 starting from the execution of a blinky to download all the needed files.

A.4.1 Build a simple "blinky" example

Follow this procedure to build a simple blinky example from the demo folder of the FreeRTOS distribution. In order to execute a general project on FreeRTOS simulator for Linux the following steps must be followed.

1. Install some common prerequisite packages: native **gcc** (the same used to build Linux applications) and **cmake**.

   ```
   sudo apt install gcc cmake
   ```

2. Download Freertos: the version used for this project is the one reported here [26].

   ```
   git clone https://github.com/FreeRTOS/FreeRTOS.git
   ```

3. Make a copy of sample: `FreeRTOS/Demo/Posix_GCC`.

   This blinky example is made of a software timer (which executes a call-back every 2 seconds) and a thread (infinite loop with a timed wait of 200 ms). Both write a single byte into a message queue on every iteration. Another thread waits for bytes to show up in the queue, recognizes the sender and writes a message for each on `stdout`.

4. Build without modification: `make` inside `FreeRTOS/Demo/Posix_GCC`

   The project is built using a Makefile: the variable `SOURCE_FILES` contains all the source files to be compiled, and the variable `INCLUDE_DIRS` is the concatenation of all the include paths to be passed to the compiler command. You may also want to edit `FREERTOS_DIR_REL` at the top of the Makefile if you move the sample to a different folder relative to the FreeRTOS distribution.

5. run: `./build/posix_demo`

   or what you specified in the variables `BIN` and `BUILD_DIR` in the Makefile.

A.4.2 Set up TAP

Set up the TAP on the device, more details here 5.2.1.

1. `sudo ip tuntap add dev tap1 mode tap user ‘whoami‘` # *the tap will be called tap1 and be property of the current user*

2. `sudo ip link set dev tap1 up`

3. `sudo ip addr add 192.168.2.1/24 dev tap1` # *the computer assigns this address to the interface*

4. `export PRECONFIGURED_TAPIF=tap1` # *Set the environment variable*

A.4.3 Build and run the TCP/IP WG project

Copy the directory **Posix_GCC_lwip2_wg** inside `FreeRTOS/Demo` directory. In order to set up the WireGuard configuration, change the needed variables inside `main_blinky.c` according to the settings of the Client and Server. Open a Terminal and run Netcat[24] opening the 3333 port:

```
nc -l -v 3333
```

Open a Terminal in the project directory (remember to set up the TAP environment variable) and build:

```
make
```

Execute:

```
./build/posix_demo
```

A.5 Server Configuration

In this section it is analysed how to configure WireGuard on the server side. It is recommended to use a Linux device or an Android device. The following instructions are for 3 different types of application:

- Linux command line
- Windows application
- Android application

A.5.1 Linux server

In order to setup WireGuard on a Linux environment it is necessary first of all to install the software running

```
sudo apt install wireguard
```

Then must be created a private and a public keys for the Wireguard server

```
wg genkey                    # generate private key
echo "<private key>" | wg pubkey    # generate public key
```

or to run both commands in the same line and write the keys on two distinct files in the current directory.

```
wg genkey | tee private.key | wg pubkey > public.key
```

At this point remains to decide the IP addresses for the tunnel. A range of IPs could be selected for the interface (i.e. 192.168.77.1/24) that will be the potential values for the available peers. Now all the required parameters are set and the wg0.conf file could be configured. In this file must be described the interface of the server and the peers (see in section A.3 to more details) with code similar to the following one:

```
[Interface]
Address = 192.168.77.1/24
ListenPort = 51820  # default
PrivateKey = {private key}

[Peer]
PublicKey = {public key}
AllowedIPs = 192.168.77.2/32
```

The values of the keys are the one generated before and the IPs of the peer must remain in the range of the interface. Go in the `/etc/wireguard` and make a new `wg0.conf` file (must be admin to access the directory).

```
sudo -i
cd /etc/wireguard
nano wg0.conf
```

Set up UFW firewall rules to open required ports:

```
sudo ufw allow 51820/udp
```

Verify it:

```
sudo ufw status
```

Turn the WireGuard service at boot time using the systemctl command, run:

```
sudo systemctl enable wg-quick@wg0
```

Start the service, execute:

```
sudo systemctl start wg-quick@wg0
```

Get the service status, run:

```
sudo systemctl status wg-quick@wg0
```

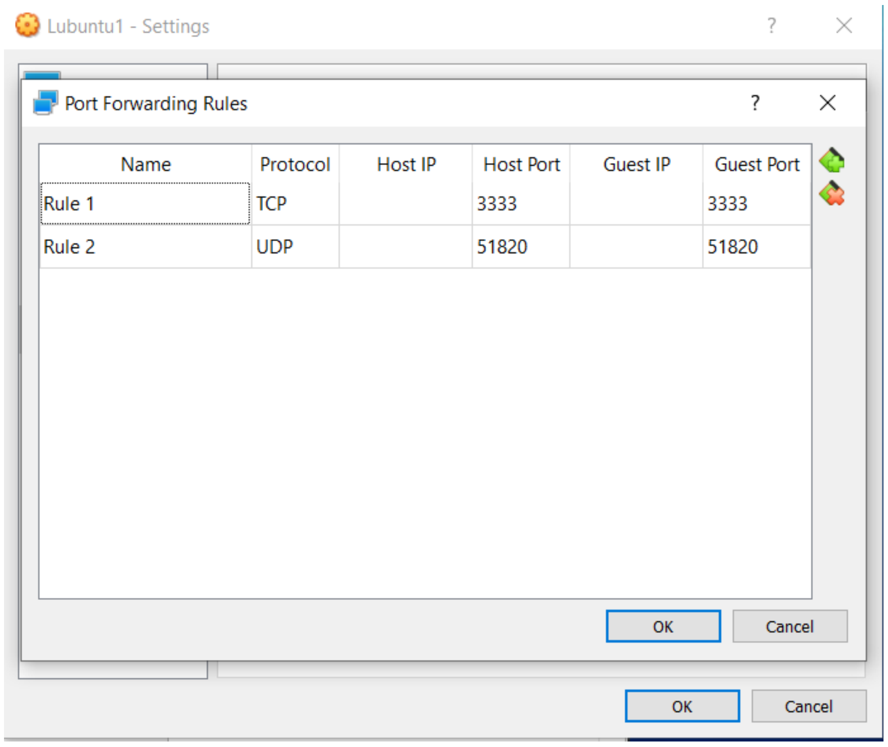
Verify that interface named wg0 is up and running using the ip command:

```
sudo wg
sudo ip a show wg0
```

Done! The WireGuard Server is working.
Remember to use the following commands to turn up or down the Virtual Network Interface

```
wg-quick up wg0
wg-quick down wg0
```

NOTE: In case of using Linux on Oracle VirtualBox VM the network settings of the virtual machine must be adjusted accordingly to server settings. Go to **Machine** $\rightarrowtail$ **Settings** $\rightarrowtail$ **Network** under Menu bar, make sure it is attached to `NAT`. Go under **Advanced** $\rightarrowtail$ **Port Forwarding** then add two new forwarding rules with the plus button. Add a rule for the TCP communication and one for the UDP. An example is shown in A.4



Name	Protocol	Host IP	Host Port	Guest IP	Guest Port
Rule 1	TCP		3333		3333
Rule 2	UDP		51820		51820

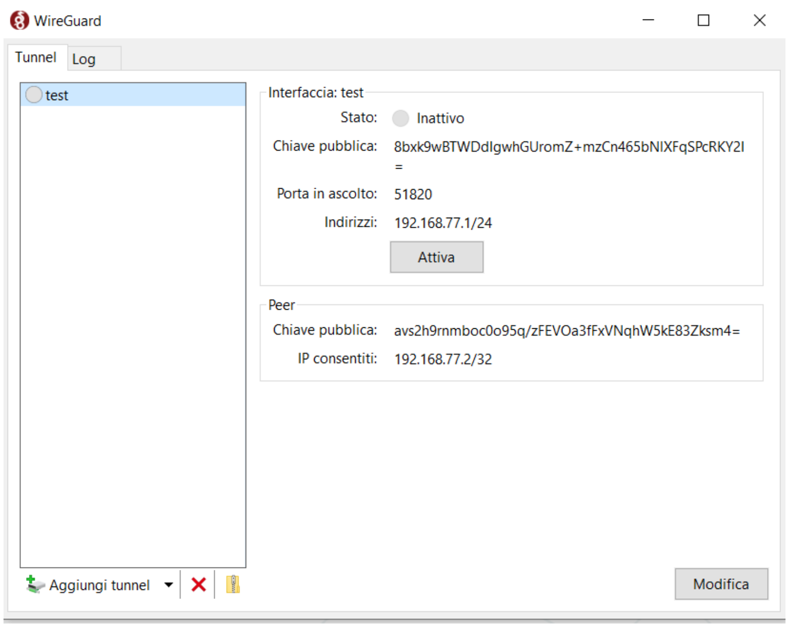
Figure A.4: Forwarding rules for Oracle VirtualBox VM.

A.5.2 Windows server

On windows it is first of all necessary to install the WireGuard tool from the following link [27]. Now you can open the GUI of the tool and create a new tunnel, using the bottom-left button (A.5), where you can create a new empty tunnel or import one from a file. in figure A.6 there is an example of how to create a new empty tunnel test where there are:

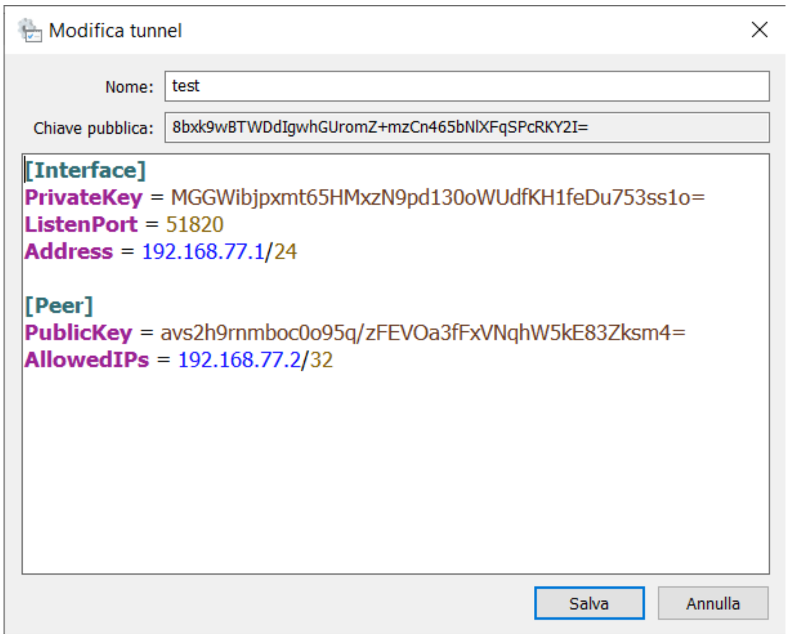
- one private interface, that specifies
  - a private key
  - the listen port of the connection
  - the address of the network interface
- one peer, represented by
  - a public key
  - a range of allowed IPs (in this case only one peer is configured)

Once the tunnel is correctly created you can activate it using the activation button, or modify the previous parameters using the bottom-right button.



```
Interfaccia: test
Stato:              Inattivo
Chiave pubblica:    8bxk9wBTWDdIgwhGUromZ+mzCn465bNIXFqSPcRKY2I
                    =
Porta in ascolto:   51820
Indirizzi:          192.168.77.1/24

Peer
Chiave pubblica:    avs2h9rnmboc0o95q/zFEVOa3fFxVNqhW5kE83Zksm4=
IP consentiti:      192.168.77.2/32
```

Figure A.5: WireGuard windows application main page.



Nome: test

Chiave pubblica: 8bxk9wBTWDdIgwhGUromZ+mzCn465bNIXFqSPcRKY2I=

```
[Interface]
PrivateKey = MGGWibjpxmt65HMxzN9pd130oWUdfKH1feDu753ss1o=
ListenPort = 51820
Address = 192.168.77.1/24

[Peer]
PublicKey = avs2h9rnmboc0o95q/zFEVOa3fFxVNqhW5kE83Zksm4=
AllowedIPs = 192.168.77.2/32
```

Figure A.6: WireGuard windows tunnel settings.

A.5.3 Android server

Also in this case the first thing to do is to download the application (from the Google Play Store). At the start-up appears the main page (left picture in A.7), then you can click on the blue button and, selecting for example ”build from scratch”, you want to create a new tunnel. Now, a page with all the required settings will appear. As explained before must be set the name, the private key and the address for the network interface. Then clicking on "add peer" a peer could be added and could be set also here the public key and the range of IPs. At this point all the setting must be saved pressing the icon in the top-right corner and, using the button shown in the figure on the far right, could be turned on the corresponding tunnel.

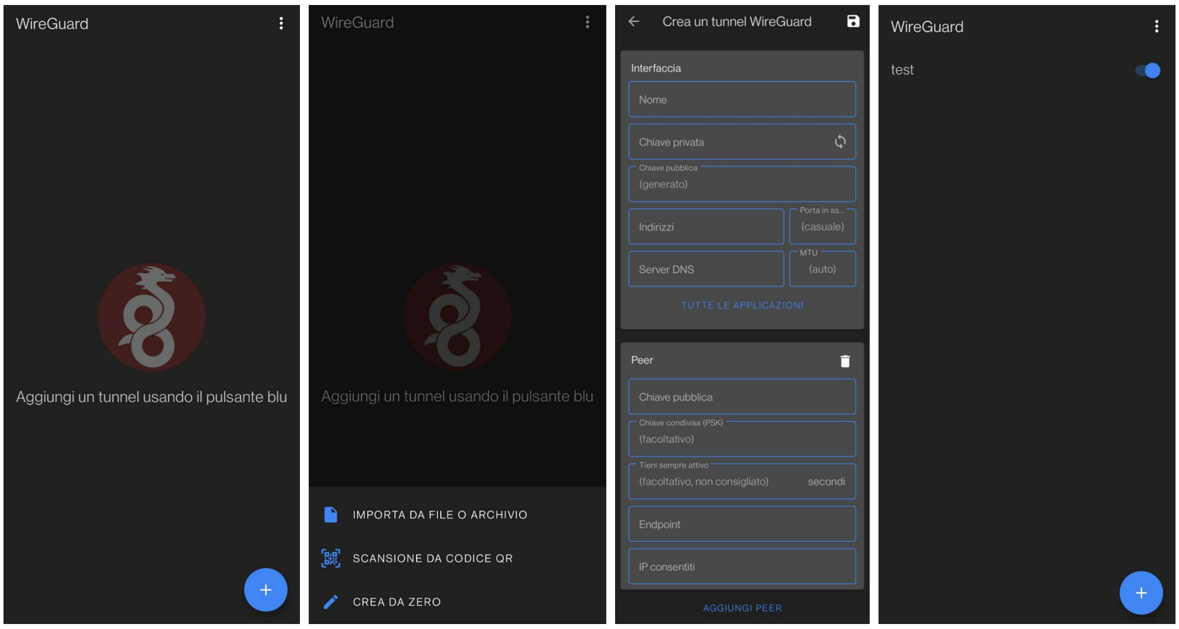


Figure A.7: WireGuard android application.

Bibliography

[1] *WireGuard white paper.* URL: https://www.wireguard.com/papers/wireguard.pdf.

[2] *FreeRTOS official page.* URL: https://freertos.org/.

[3] *FreeRTOS official tutorial guide.* URL: https://www.freertos.org/fr-content-src/uploads/2018/07/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf.

[4] *LwIP port to an operating system.* URL: https://lwip.fandom.com/wiki/Porting_for_an_OS.

[5] *LwIP NO_SYS documentation.* URL: http://www.nongnu.org/lwip/2_1_x/group__lwip__opts__nosys.html.

[6] *LwIP port to NO_SYS==1 (without an operating system).* URL: https://lwip.fandom.com/wiki/Porting_For_Bare_Metal.

[7] *LwIP RAW API.* URL: https://lwip.fandom.com/wiki/Raw/native_API.

[8] *LwIP netconn API.* URL: https://lwip.fandom.com/wiki/Netconn_API.

[9] *LwIP BSD socket API.* URL: https://lwip.fandom.com/wiki/Socket_API.

[10] *ESP-IDF official page.* URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/index.html.

[11] *wireguard-lwip module esp32 source.* URL: https://github.com/trombik/esp_wireguard.

[12] *wireguard-lwip module original source.* URL: https://github.com/smartalock/wireguard-lwip.

[13] *other wireguard-lwip project tried.* URL: https://github.com/zmeiresearch/wireguard-lwip.

[14] *ESP-IDF Add a component and buil details.* URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/build-system.html.

[15] *LwIP 2.1.3 source download link.* URL: http://download.savannah.nongnu.org/releases/lwip/lwip-2.1.3.zip.

[16] *LwIP contrib package 2.1.0 download link.* URL: http://download.savannah.nongnu.org/releases/lwip/contrib-2.1.0.zip.

[17] *TAI64 timestamps.* URL: https://cr.yp.to/libtai/tai64.html.

[18] *WireGuard protocol overview, secion about mitigation of DoS situations / attacks.* URL: https://www.wireguard.com/protocol/#dos-mitigation.

[19] *WireShark official page.* URL: https://www.wireshark.org/.

[20] *Virtual Box VM.* URL: https://www.virtualbox.org/.

[21] *Ubuntu Linux.* URL: https://www.ubuntu-it.org/.

[22] *Standard Toolchain Setup for Linux and macOS.* URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/linux-macos-setup.html.

[23] *Github ESP-IDF*. URL: https://github.com/espressif/esp-idf.git.

[24] *Netcat commands*. URL: https://linuxize.com/post/netcat-nc-command-with-examples/.

[25] *Netcat app for Android*. URL: https://play.google.com/store/apps/details?id=com.werebug.androidnetcat&hl=it&gl=US.

[26] *FreeRTOS-202112 Distribution Download*. URL: https://github.com/FreeRTOS/FreeRTOS/releases/download/202112.00/FreeRTOSv202112.00.zip.

[27] *WireGuard download page*. URL: https://www.wireguard.com/install/.