UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona

Design of a Secure Transmission System for Secure Key Injection During Initialization Phase of IOT Devices

A Master's Thesis

Submitted to the Faculty of the

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

by

Rafael de la Rosa Ortiz

In partial fulfilment

of the requirements for the degree of

MASTER IN TELECOMMUNICATIONS ENGINEERING

Advisor: Francisco José Rico Novella

Barcelona, January 2020

Title of the thesis: Design of a Secure Transmission System for Secure Key Injection During Initialization Phase of IOT Devices

Author: Rafael de la Rosa Ortiz

Advisor: Francisco José Rico Novella

Abstract

In the last decade society has experienced an exponential growth in the number of devices connected to the Internet. Recently, new gadgets called Internet of Things devices have appeared in our homes. Although they often lack a physical interface to directly interact with them, they are able to read information from sensors and autonomously communicate with servers, performing decisions accordingly.

However, most of the domestic devices that are being commercialized do not implement strict security policies, potentially leading to security breaches that compromise the user's privacy.

The following work provides an alternative to the WPS technology in the initial setup phase of these devices, in which the gadget has to be loaded with the Wi-Fi key so it can connect to the Internet.

The use of infrared technology implementing a Diffie-Hellman key exchange protocol to inject this key makes the process much safer, without compromising the cost of the device or the user experience.

To my family, and to my friends during my stay in Barcelona coursing the master's degree, for both supporting and bearing me during these two years.

Acknowledgements

In first place, I would like to thank my family, for granting me the possibility of study the master's degree in UPC Barcelona, and also for their constant support and encouragement.

My completion of this project could not have been accomplished without the constant assistance of the thesis director, Francisco Rico, who did not only have the initial idea for the project, but also guided me throughout it, having always time to solve any doubt I could have, helping me with the hardware part and continuously suggesting new ideas about the development and the design. For all of this, I would like to express my sincere gratitude to him.

A special word should also be headed towards my friends in the residence where I stayed in Barcelona during my studies. Their constant support, and all the spare time hanging out with them have also helped me in the completion of this work.

Finally, I want to thank my university friends, my professors and all the people I have met during these two years that, somehow, have had some kind of influence on me in this stage.

Revision History and Approval Record

Revision	Date	Purpose
0	08/15/2019	Document creation
1	09/30/2019	Document first preliminary version
2	10/31/2019	Document second version. Content Completed
3	12/19/2019	Document final version
4	01/06/2020	Last Changes Approved by Project Supervisor

Written by:		Reviewed and approved by:	
Date	01/06/2020	Date	01/06/2020
Name	Rafael de la Rosa Ortiz	Name	Francisco José Rico Novella
Position	Project Author	Position	Project Supervisor

Table of Content

Abstract 1
Acknowledgements 3
Revision History and Approval Record 4
Table of Content 5
List of Figures 7
List of Tables 8

1. Introduction 9
- **1.1. Problem and Scenario Description** 9
- **1.2. Justification of the Solution Adopted** 11

2. State of Art 13
- **2.1. Key Distribution in IoT Devices** 13
- **2.2. Wi-Fi Vulnerabilities** 15
  - 2.2.1. Key Reinstallation Attack 15
  - 2.2.2. Wi-Fi Protected Setup Attack 15
- **2.3. Bluetooth Vulnerabilities** 17
  - 2.3.1. Elliptic Curve Checking in Bluetooth Key Exchange 17
  - 2.3.2. BlueBorne Set of Vulnerabilities 17
- **2.4. Privacy in IoT Domestic Environments** 18
  - 2.4.1. Privacy Control within a TOR Distributed Network 20
  - 2.4.2. Vulnerability Discovery of IoT Devices Using Shodan Search Engine 22

3. Project Development 23
- **3.1. Scenario Architecture and Configuration** 24
- **3.2. Hardware** 28
  - 3.2.1. ESP8266 NodeMCU as IoT Device 28
  - 3.2.2. ESP8266 Heltec Wi-Fi Kit 8 as IoT Controller 32
  - 3.2.3. Raspberry Pi Model 3b+ as the IoT Hub 33
- **3.3. Software** 35
  - 3.3.1. ESP8266 36
  - 3.3.2. Raspberry Pi 60
- **3.4. Developmental Setbacks** 66

4. Results 68
- **4.1. Hardware Adopted** 68
- **4.2. Software Adopted** 69
- **4.3. Private Key Transmission Results** 70
- **4.4. Key Loading into the IoT Router** 71

5. Future Development and Conclusions 72
- **5.1. Future Development** 72
- **5.2. Conclusions** 75

6. Budget 76

Bibliography 78
Annex 80
Annex I - ESP8266 Controller Code 80

Annex II - ESP8266 Receiver Code 104
Annex III - RaspPi WIFI_SETUP.PY Code 122
Annex IV - RaspPi CAMERA_SCAN.PY Code 125
Annex V - RaspPi SERIAL_READER.PY Code 127
Glossary 128

List of Figures

Figure 1 - Risk Assessment Matrix 10
Figure 2 - User Connecting to a TOR Network 20
Figure 3 - ShoVAT Shodan Based Modular Architecture 22
Figure 4 - IoT Network Topology Proposal Diagram 24
Figure 5 - Raspberry Pi Model 3b+ Board 25
Figure 6 - ESP8266 NodeMCU Board 28
Figure 7 - IR LED Transmitter Circuit 1 LED 29
Figure 8 - IR LED Transmitter Circuit 2 LEDs 30
Figure 9 - IR Receiver Circuit 31
Figure 10 - Heltec Wi-Fi Kit 8 Board 32
Figure 11 - Raspberry Pi Model 3b+ Ports and PINs 33
Figure 13 - IR Protocol Transmission Summary Workflow 51
Figure 14 - 3rd Working Version Transmission Detailed Diagram 52
Figure 15 - 4th (Current) Working Version Transmission Detailed Diagram 54
Figure 16 - 2nd Version Time Diagrams | Transmission in Clear 56
Figure 17 - 3rd Version Time Diagrams | Clear & Encrypted Transmission 57
Figure 18 - 4th (Current) Version Time Diagrams | Clear & Encrypted Transmission 58
Figure 19 - Transmission Time Evolution with Version 59
Figure 20 - QR Scanner Script State Diagram 63
Figure 21 – Access Point Setup Alternatives in RaspPi 65

List of Tables

Table 1 - Error Probability 1 LED vs 2 LEDs....30
Table 2 - QR Code Version, Size and Capacity Comparison....38
Table 3 - Controller EEPROM Memory Mapping....39
Table 4 - Receiver EEPROM Memory Mapping....41
Table 5 - ACK Frame Structure 16 bits....46
Table 6 - HELLO Frame Structure 36 bits....47
Table 7 - DATA Frame Structure 36 bits....48
Table 8 - XOR Logic Table for Encryption....50
Table 9 - Hardware Cost Analysis....76
Table 10 - Dedicated Hours Analysis....77

1. Introduction

1.1. Problem and Scenario Description

The main problem this project pretends to solve is the lack of security and privacy when configuring a brand new IoT device by an end user, who might not have any technological knowledge.

In this situation, is hard to design an easy-to-use and secure mechanism to authenticate the IoT devices without compromising their security, or even the overall wireless network protection. In most of the cases, we are working with tiny gadgets which may lack of any sort of input peripheral, screen or any other sort of device to physically input or retrieve a password or any kind of information.

Therefore, most manufacturers rely on Wi-Fi Protected Setup (WPS) technology to establish the connection between the access point of a domestic router and the device itself[1b]. Consequently, this procedure may seem to be reasonable at first, as it is offering a practical solution to the aforementioned scenario solving the obstacle of the absence of input mechanisms. WPS technology only requires the IoT device to implement a single input interface: a button acting as a WPS connector.

WPS technology mainly employ (simultaneously or exclusively) two different mechanisms to connect a device to the access point:

- **Using a PIN**. In this case, the device must have any kind of inputting interface to enter the PIN and connect to the router. This solution is commonly used with mobiles and laptops, as in these cases entering a PIN should not be a problem.
- **Pushing router's WPS button** while ordering the device to find and connect with the recently WPS-opened router inbounds.

Between the two options above, we are interested in the latter, as it is the one adopted by most of the manufacturers of IoT devices. The final user only needs to have physical access to both the IoT gadget and the router, which is not a problem in most of the cases.

Connection between access point and IoT device will be achieved after pushing the router's WPS button and, within a small period (generally between 30 and 60 seconds), ordering the IoT device to connect to the router (for instance, by pushing the IoT node's button acting as a WPS connector as mentioned before).

Once this process is completed, the device has received the WPA Key and it is able to authenticate itself to the router. However, this process is not guaranteed to be secure due to the small window of opportunity in which any person can connect to the router, as well as the vulnerabilities the WPS protocol has been suffering since its massive worldwide adoption.

At the end of the configuration phase, no one can guarantee that the domestic network has not been compromised, as any outsider could have sniffed the secret key, becoming part of the network. Moreover, most users do not have the enough knowledge to detect and identify the presence of third persons in their domestic network, aggravating the consequences of the use of this insecure implementation.

Therefore, there is no need for the attacker to have physical access to the router or to any of the devices for this scenario to occur. The only requirements are to be relatively close (within ~20 meters) and to be in possession of the necessary equipment to audit the wireless network state and security.

As both necessary conditions are easy to achieve, if we make a Risk Assessment[2b] analysis for the described scenario, we can conclude that the risk or probability of this to happen is high (around 4/5 in the used scale) and the impact of it when it has happened, the severity, is very high or catastrophic (a 4/4 in the scale). If we link both factors through a Risk Assessment Matrix, like the one below, we can conclude that the Risk we are trying to reduce is currently very high.

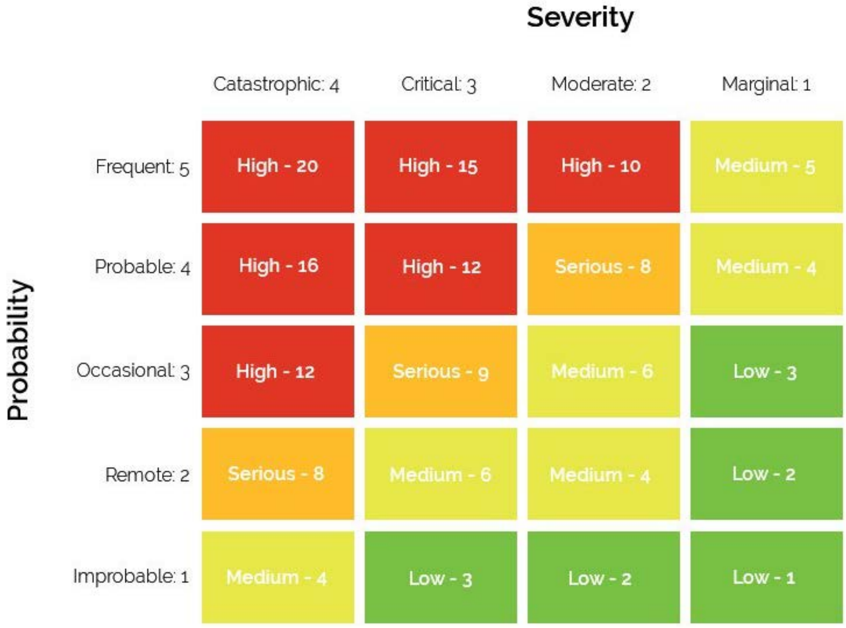

Figure 1 - Risk Assessment Matrix

Despite not being able to reduce the impact directly, we can work on reducing the risk of this occurring. This can be done by designing a more secure procedure to initialize these devices, at the same time we try to keep the process as much user-friendly as possible. In the end, commercial viability of a solution requires easiness of use by the end user.

1.2. Justification of the Solution Adopted

The primary goal of this essay is to offer an alternative in the configuration phase of any commercial IoT device. As stated before, the vast majority of the most sold solutions implemented by most manufacturers to load or inject the secret key into an unconfigured IoT device do not consider important factors such as security and privacy.

Key Injection is the most critical phase in the configuration process, as this secret key will be used later to authenticate the device to an Access Point, as the first and essential step before establishing a wireless communication with it, accessing internet, and allowing the user to configure and customize the device remotely.

Therefore, before jumping into the core parts of the solution, it is important to specify a few assumptions that were made in the process of its design:

- The IoT devices are kept physically safe. Third parties do not have direct access to them.
- After the key exchange, it is supposed that all parties in the IoT ecosystem are safely storing the keys in their memory, and they cannot be retrieved by force.
- Routers use a secure and not-vulnerable key encryption protocol, such as WPA2.
- Not every user can afford updating their routers, therefore, each router may implement different WPS versions and not all of them are guaranteed to be secure and updated.

Despite WPS security has greatly improved in the recent years, technology is under constant evolution and it is very likely that new vulnerabilities and design inefficiencies will come up from time to time. The following solution tries to put an end to this situation, with the certainty that it will remain secure during a long time, almost eliminating the possibilities and risks of suffering a successful attack.

As a summary, our solution is a direct alternative to what WPS currently offers. Current method is completely sacrificing security in terms of usability. Our model implements a much more secure alternative at the cost of slightly increasing usability complexity.

In the end, when a decision between usability and security must be taken by a final user, in most of the times the former is chosen. This is due to lack of awareness in security by most of the people, who will ultimately choose the easiest and fastest path without taking other factors into consideration.

As a result, most manufacturers exploit this fact by designing and marketizing fastest solutions without investing in security and privacy. The proposed mechanism already implements robust security features by default, besides being easy to use by non-experts.

Also, the proposal is not only limited to the transmission of the private key itself, but it also covers other important steps of the initial configuration process, such as a true random key generation methodology using a high entropy level source, as well as the interface and interaction with the final user, allowing him in last instance to read and use the generated key.

Finally, an architecture based on privacy and user control is proposed. It allows advanced users to configure and monitor the connections his IoT devices establish with external services via internet. It will provide them with advanced tools to choose which services as servers each device is able to connect to, preventing them from accessing unwanted and unplanned services that may not be of any interest and might compromise privacy.

2. State of Art

2.1. Key Distribution in IoT Devices

Today, most of commercial IoT devices include a default private key injected in their firmware during the manufacturing process[3b]. To protect this key and the device itself, usually the firmware is encrypted and securely loaded via a secure boot loader. However, this does not guarantee the security of the device, as instead of compromising the firmware, which is encrypted, the boot loader can be targeted. No matter how many security encrypted layers we add, as in the bottom of them there always must be a plaintext-programmed script to boot and load the firmware and the associated private key.

Therefore, factory injected keys are vulnerable during the booting phase. Moreover, most of these keys are usually designed to be temporal and manufacturers recommend changing them as soon as we perform the initial device setup. This recommendation is frequently ignored by users, as it would involve investing additional time in the configuration, and, from their point of view, the predefined password works perfectly fine.

Today, in domestic IoT, the vast majority of the industry employs Wi-Fi or Bluetooth technologies for distributing a new randomly generated symmetric private key between the domestic IoT hub controller (a mobile phone, a computer or a specific IoT device) and the different IoT local nodes (lights, speakers, TVs…) which are also usually connected to the router via Wi-Fi.

Moreover, many IoT devices does not include a wired connection port such as Ethernet or USB, meaning that their configuration has to be performed wirelessly.

This initial setup assumes two important facts:

i) The Wi-Fi network or Bluetooth connection is completely secure, meaning that no third person is eavesdropping on our traffic at the time of the IoT nodes setup or key injection.

ii) The Central IoT Node is able to generate truly random numbers, as the base for creating a private key that will be injected in the different client nodes.

Both statements cannot be confirmed in every situation. Wi-Fi technology has been proven over the time to be unsecure or, at least, not as safe as it claims itself to be. In fact, a large number of vulnerabilities has been detected in the last decade, such as the well-known Wi-Fi Protected Setup (WPS) exploit or the Key Reinstallation Attack, both compromising WPA2 security. Moreover, even if we someday manage to create a robust and secure Wi-Fi network, there will always exist other possible unavoidable point of failures, such as human errors when configuring the network or establishing its associated security key.

If an attacker is connected to the Wi-Fi network in the moment of the key injection[4b], he will be able to intercept and decrypt it, gaining control ultimately over all of our IoT devices, which may lead to terrible consequences in function of the kind of devices we were configuring.

Once all of the previous information is considered, there could be a solution for this aspect: using a Diffie-Hellman key exchange approach[1]. This method was specifically design for this purpose: exchanging keys between devices, guaranteeing that no one will be able to read the information exchanged, assuring the confidentiality of the private key.

However, Diffie-Hellman does not protect against a combined Interception, Interruption and Fabrication attack, as it is discussed later in the present document.

Also, it is really important to assure that there is a device with enough source of entropy as to support Real Number Generation (RNG). Otherwise, an attacker could discover potential exploitable patterns in the private key generation, easing the task to crack the private key.

In definitive, it can be concluded that today's private IoT key management and injection in a domestic IoT ecosystem is insecure, as some of the assumptions considered when designing the IoT node device setup and the private key injection are not always met.

2.2. Wi-Fi Vulnerabilities

WPA2 is not as secured as it is claimed to be. There are several attacks and exploits able to expose the traffic of an encrypted Wi-Fi network. In this document two of them are explained due to the fact they could drastically compromise the injection of a private IoT or WPA key.

2.2.1. Key Reinstallation Attack

In the middle of 2017, a vulnerability regarding an exploit affecting the 4-way handshake procedure used in WPA2 was publicly exposed. This attack, called Key Reinstallation Attack (KRACK), was firstly described in an investigation paper[2].

This threat allows an attacker to force reset the nonce of the encryption key used by the client to authenticate to the Router. This fact ultimately allows the attacker to decrypt many of the packets, by resetting some key parameters of the randomization algorithm used beside the encryption key itself. Two common parameters used in this randomization are the nonce, which represents the number of transmitted packets, and the replay counter, which reflects the number of received packets.

The exploit allows to decrypt the packets mainly travelling from the client to the router (although packets from router to client can be sniffed if we initially target the router). A critical issue regarding this attack is that every device has to be patched individually via a software update. This means that updating the router does not prevent a device connected to it from being attacked, and vice versa.

Despite not revealing the WPA2 key itself, in our scenario this attack would allow the attacker to intercept the private key sent by a user to an IoT device in the setup process. The risk of exposure to this attack gets multiplied if we considered the fact that many people don’t update their devices, so it can lead to a potential exposure of the information travelling through a Wi-Fi connection.

2.2.2. Wi-Fi Protected Setup Attack

Wi-Fi Protected Setup (WPS) is a mechanism certified by the Wi-Fi Alliance and implemented in most of modern routers which allows for a quicker an easier way of connecting a device to the router, as an alternative to the classical way of manually writing the password in every device. Non-specialists people often use this method rather than manually entering the key due to its easiness of use.

However, WPS has proven to be a concerning security hole. Most security organizations[5b] have agreed it is better to not implement it or to disable it if possible.

As Indra Dwi Rianto[3] clearly explains, WPS can be implemented as a push button to enable pairing between a device and the router and/or an 8-digit numeric PIN. The key issue about this Brute-Force attack is that it is not needed to guess the 8 digits all together.

Once the first 4 digits are found, the router will return a checkpoint confirming they were correct, letting the attacker know when to move on to the last 4 digits. Brute forcing two groups of 4 digits each is far easier than guessing them all together. Also, the 8th digit of the PIN is just a checksum, so in the end the guess is reduced to 4+3 digits. If the first group of 4 digits and the second group of 3 digits are guessed, it is trivial to calculate the checksum that represents the 8th digit. In practice, we just need 11.000 (10^4+10^3) attempts to crack the PIN, which can be easily done in a few seconds.

There are many available tools in the website that automatize this attack, for investigation purposes. One of the best known is called Reaver. There are many research papers[4] which describe how to conduct one of the possible variants of the WPS Brute Force attacks.

There are two ways of protecting against this attack: disabling the WPS option in the router configuration software or using a router without WPS support. Also, most of the newer routers come with an effective way of blocking an attacker against a several number of failed attempts. Nonetheless, most manufacturers and modern routers are designed to detect and stop answering to WPS requests if they detect a brute force guess, making it harder nowadays to apply this technique.

Once the PIN has been cracked, the attacker is sent the encryption key (WPA, WPA2…) the router uses, becoming the intruder part of the network. This would allow him to perform many different attacks, including retrieving any kind of confidential information like the private IoT key if the owner of the network is configuring his IoT devices using Wi-Fi.

2.3. Bluetooth Vulnerabilities

Bluetooth can be considered as a much safer protocol than Wi-Fi to interconnect devices. However, it is not free of security vulnerabilities[5]. Most of them usually affect specific devices, although some others are related to the protocol itself. Some of those regarding the second mentioned group are briefly exposed in this document as an example.

2.3.1. Elliptic Curve Checking in Bluetooth Key Exchange

One of these vulnerabilities is the VU#304725[6b] discovered in the middle of the 2018. It states that the part of the latest Bluetooth firmware in charge of validating the elliptic curve parameters during Diffie-Hellman key exchange.

This situation may result in a remote attacker being able to utilize a man-in-the-middle attack and to determine the cryptographic keys if performed during the key exchange.

This breach can be solved by updating each device firmware, and many of the compromised firmware version are already substituted by patched ones. However, if a device is not patched, someone could intercept his Bluetooth pairing key and read the traffic it exchanges, including a cryptographic key if the victim was performing the setup of an IoT device using Bluetooth.

2.3.2. BlueBorne Set of Vulnerabilities

A set of vulnerabilities, known as BlueBorne[7b], was discovered in 2017 affecting many different devices and operative systems supporting Bluetooth connectivity.

Each of those vulnerabilities affects specific functions of the protocol in specific devices. Some are related to causing overflows in the victim, reading or writing in forbidden memory regions, etc. Also, it may allow an attacker to perform a Man-in-the-Middle attack with all the consequences that entails.

As a result, an attacker could obtain private information, including a private IoT key if the victim is performing the setup of an IoT device using Bluetooth. It could even allow an attacker to remotely execute code in a victim's device.

As it is a firmware issue in all the cases, they can be solved just by patching it via software updates. However, is an important issue to consider as most people are not up to date with their device's updates.

2.4. Privacy in IoT Domestic Environments

IoT security and privacy are both concepts frequently applied to any given IoT application, constituting a single domain problem[6]. However, this approach should be avoided as domestic IoT needs, challenges and objectives usually differ from the general approach. The needs of a critical wide infrastructure or a commercial operation are different from the ones that a domestic device may have. The financial and human resources used to operate or manage them are also different. Human factor, for instance, is much more influential and critical in a domestic ecosystem. Therefore, it would be convenient to keep domestic IoT at a different domain of application and needs.

Apart from security, privacy is another major concern in any IoT network. This project introduces some ideas and designs to enhance the privacy of a domestic IoT ecosystem, which will be covered later.

Despite its importance, this factor is often not covered in depth by manufacturers and designers, leading to potential issues and loss of privacy for the users of these type of devices. In the last years, many researchers and security experts have been working in this field, and many research papers have been published analyzing possible case scenarios and privacy risks, as well as providing some solutions to them.

IoT devices need connectivity to Internet to be able to perform their functions. Some of them allow the user to remotely manage and interact with them, while others need to periodically query external servers for some kind of information. These servers often gather all kind of sensitive and private data from our devices and could, in last instance, send malicious information or try to take over the device[7].

Depending on the device, different type of sensible information is retrieved by the servers the device interacts with. For example, an intelligent light bulb that sends data statistics could end up exposing the user schedules and routine. With the daily information about the periods in which the light bulb is on and off, one could make a precise estimation about the times the user is at home or away.

Other major concern is the impossibility for a user to know what kind of information the device is really gathering. One clear example for this case is the hidden microphones in some devices that do not really use it for their functions. Earlier this year, a smart cuisine mixer robot, commercialized by Lidl in some European countries, was found to incorporate a hidden microphone that might be listening to private conversations of their users[8b]. This fact has cause major controversy, as the existence of this component was never advertised in the product specifications.

Analogously, the act of connecting to an Internet service reveals the IP address of the device connecting to it. In a close future, large information servers will raise as the demand increases to answer the need of information for sensors and IoT device. They will act as an updated library that provide the answer to all those different IoT devices asking for an information update. However, the different is that instead of focusing on a specific field or service (for example, providing updated weather report information), they will try to comprise as many different services and information sources as they can. In the end, most IoT devices will end up connecting to one of this few megaservers, and they will make their profit from our information and big data information extraction tools.

Despite these services will show useful for us as users, they will collect millions of gigabytes of information about our habits, preferences, routines and even political opinions.

They will also be able to link each IP address with a schema of all the IoT devices connected to internet and obtain even more information. As a consequence, once all the separated information from every sensor and device is put together by a common IP, a complete radiography of a given family will be exposed and commercialized in the form of useful data for advertisers and third parties that may find it profitable.

As an attempt to prevent this privacy concern in a close IoT connected future, we provide some ideas to anonymize the traffic generated from IoT devices as much as possible, so it is no longer possible to group different IoT devices by their source IP address. The idea consists in using an internal IoT Router inside the domestic private network. This router would have advance firewall functionalities to prevent devices from connected to unwanted servers. Moreover, all the traffic coming out from this internal router would be redirected by the main home router to a TOR network, assuring its anonymity.

A TOR network[8, 9b] is a distributed communication internet network in which the traffic is randomly conducted through a random number of nodes until finally being redirected to the server it was initially mean to reach. Although this makes the connection extremely inefficient in terms of performance (speed and delay), it makes the requester node almost impossible to unveil, assuring its anonymity to the server. When someone uses TOR, he can also act as an intermediate node and redirect other people traffic towards random nodes.

As IoT devices do not require speed or low latency connectivity in most of the cases, this anonymity network could be the solution to some of these privacy issues. If the server which provides the service is not able to identify the place from which the request is coming, all the information in the request loses almost all of its value as a commercial item.

Also, some IoT devices often perform specific requests may unveil important information. For example, a weather reporter device may poll the server for a weather report about its region. There are some ways to make this not obvious for the server. For example, instead of asking for the weather report today in Madrid, we could request the weather report in all the European capitals, and this way the server will not be able to determine where we live.

Also, there are available search engines in the internet that are able to find and index online IoT devices the same way Google does with websites. The most popular website for this purpose is shodan.io[9, 10b], which frequently indexes many vulnerable devices whose owners have not changed the default access credentials, becoming accessible for anyone worldwide.

2.4.1. Privacy Control within a TOR Distributed Network

The way TOR operates makes it harder for a third party to compromise not only the integrity and confidentiality of the data, but also the real identity of the sender and the receiver of the information that is being sent.

In short, this network acts as an intermediary through which all the information should be sent. The way the data is treated across this network makes the recipient unaware whether the packet was sent using TOR or not.

In order to be able to establish a TOR session and send information through it, the user must install its client software in his communication device. This software includes a TOR directory fetcher, which is necessary to be able to acquire the list of available nodes worldwide.

Once the client knows the available nodes, he can configure some connection parameters, such as the number of intermediary TOR nodes that will handle the traffic. The larger the number of nodes, the more secure the connection becomes, although latency increases as well.

The packets within the TOR network (i.e. any packet that is exchanged between any pair of TOR nodes) are sent in an encrypted format with a fixed package length. Packets in TOR are called cells, and their length is 498 bytes. The fact of using a unique length for any given type of communication protocol makes it impossible to retrieve the type of communication that takes place.

The whole process of using TOR to communicate to a public internet server is summarized in the following picture (both the picture and its explanation have been extracted from [10]):

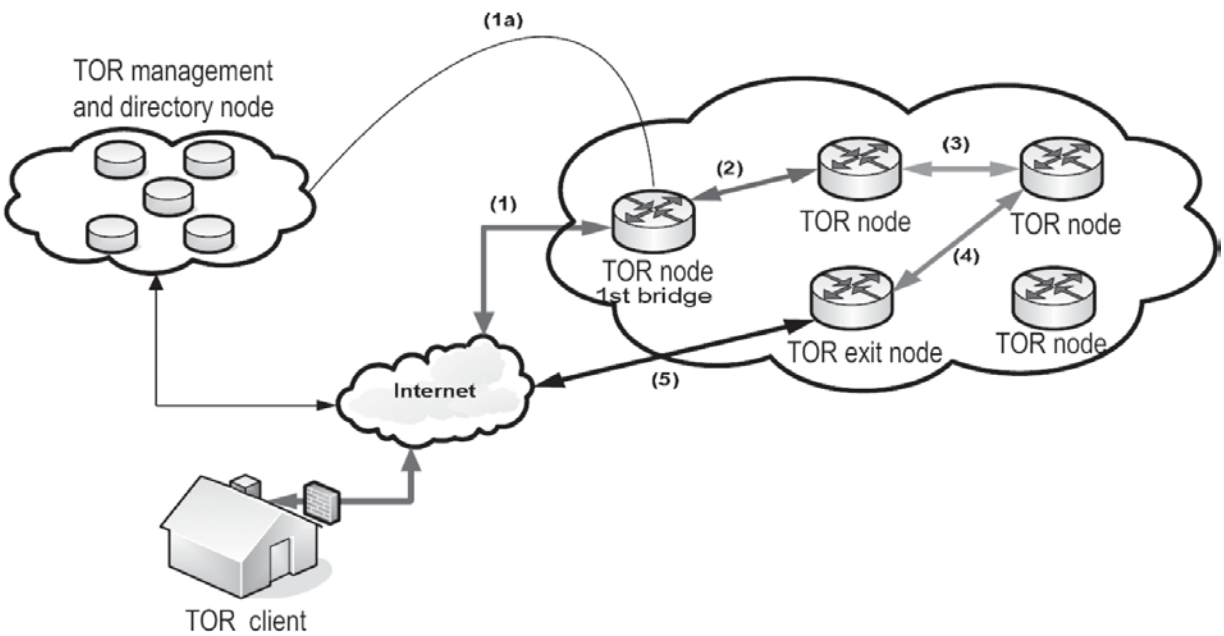

Figure 2 - User Connecting to a TOR Network

1. The user connects to the TOR network via Internet using the TOR software. It establishes a special handshake with the TOR bridge (the first TOR node).
2. The client queries the TOR management directory for the available nodes and selects some of them to establish a unique circuit cell formed by other TOR nodes worldwide that will route the communication. Each TOR node only knows his two neighbor nodes in a circuit.
3. The packets are sent to any interior TOR node in the fixed-length and encrypted cell format
4. The exit node transforms the TOR packets into standard ones in order to be able to communicate with the server using Internet protocols. Once he received a reply, he routes it back towards the first node in the circuit using the same mechanics.

2.4.2. Vulnerability Discovery of IoT Devices Using Shodan Search Engine

Shodan is a search engine launched in 2009 and designed by John Matherly. Shodan is one of the most used tools when looking for IoT devices, such as IP cameras or home automation systems, and even industrial automation hardware.

In a similar way to how website search engines work, Shodan crawls the Internet looking for online services and devices and their running ports. For each result, it locally creates a database entry consisting of its IP address, port and service banner.

When a user navigates Shodan website a performs a search, he actually queries this database. In fact, the database is updated periodically, and it can be publicly accessed using their website or Shodan API.

Shodan itself is a very powerful tool to discover services and potential vulnerabilities related to them. However, advanced search using this tool can become complex and are usually only performed by professional security researchers.

The potential of Shodan has also been expanded by many different projects from security researchers worldwide. One of the most interesting of them is ShoVAT[11].

ShoVAT was designed to improve the efficiency of researchers in this type of scenario, in which they have to generate a vulnerability report that includes a list of potential vulnerable devices, including a record of the situation of these devices during each update and software version.

It is able not only to find a vulnerable device, but also to discover when it became vulnerable and for how long it remained in this state until being patched.

It also features a vulnerability assessment tool architecture, which relies on Shodan web servers to build on top a modular architecture with many additional features which help researchers, as featured in the following figure extracted from ShoVAT paper:

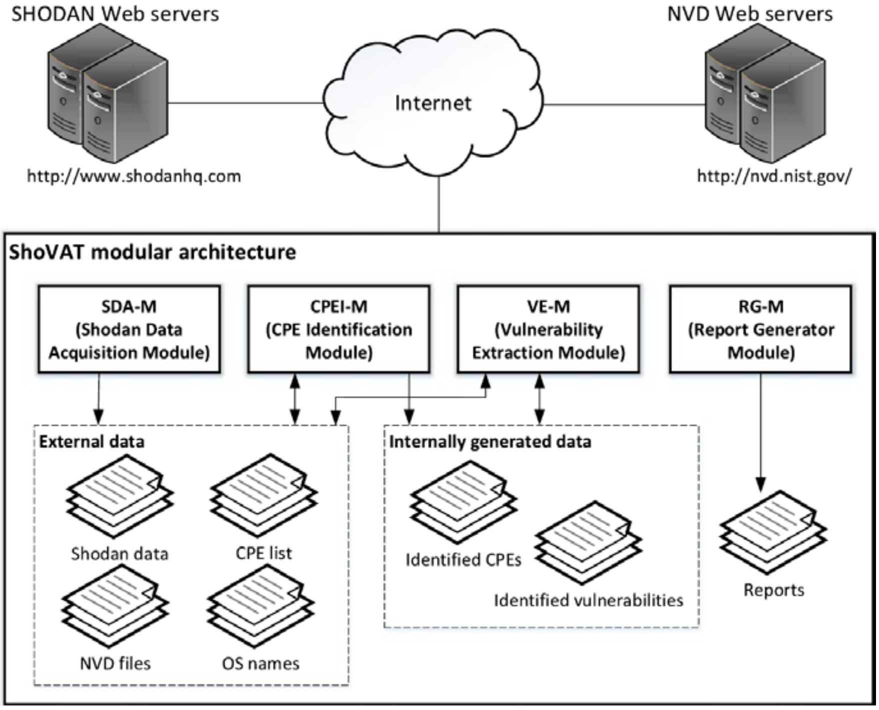

Figure 3 - ShoVAT Shodan Based Modular Architecture

3. Project Development

In this section it is covered in detail all the technical aspects of the proposed solution, the process involved in the design and construction of the solution, as well as other minor details that might be of interest and utility. It is divided into three large blocks:

- **Architecture**: Topology of the domestic network and how each device is related to the others within it.
- **Hardware**: Justification and description of the devices and electronic components used in the design of the solution.
- **Software**: Everything related with the logic and protocols implemented to allow a secure and reliable transmission using infrared technology.

The core of the project is the hardware and software parts. However, due to the time dedication and requirements for each one of these parts, it would be convenient to state that the latter was the most critical and complex of them all.

While hardware reunites several devices and components, many of them were already pre-assembled boards ready to be used, reducing the complexity of our design. However, some schematics for amplifying the power of some IR components were still needed. Otherwise, the transmission would perform erratically as the LEDs could not transmit at full efficiency, increasing the Bit Error Ratio and the retransmission and, in consequence, the overall transmission time, as well as reducing the valid transmission range.

Software part was focused on designing a reliable transmission system based on infrared connectivity. An IR library was used as the starting point, although it ended up being customized to increase its efficiency and allow bidirectional communication of large amounts of data. There are many other functionalities included in the code, memory management, use of EEPROM to store the credentials in case of power outage, management of LED display in the controller case, or an automated connection with the access point, so the user can later connect to it inside his domestic network.

There is also an additional paragraph that dives into the different difficulties, setbacks and issues which had to be overcome. The objective is to give the reader a list of details and circumstances that ended up transforming the project and significantly increasing its complexity.

3.1. Scenario Architecture and Configuration

The design proposal for our IoT domestic network fulfils two main requirements: security and privacy. Both elements are key factors in a world where everything will end up being connected to internet and, therefore, accessible by unwanted third parties. The idea is implementing additional physical layers of security to increase the isolation of our network to external agents.

The following Diagram represents the proposal for an IoT secure domestic network:

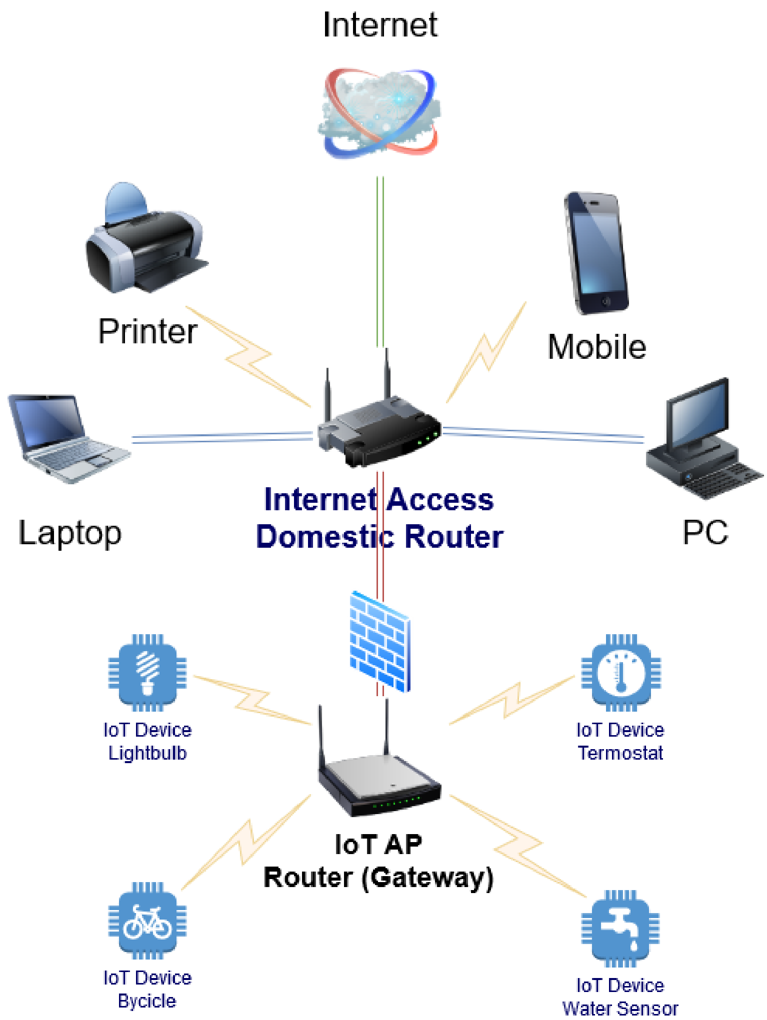

Figure 4 - IoT Network Topology Proposal Diagram

The network will feature two routers:

- **Main Router**: The one provided by the ISP. It is the responsible for providing internet connection. All non-IoT devices, such as phones, PCs or Laptops, will be connected directly to it (by wired or wireless communication). This will allow them to have full internet access with the standard restrictions imposed to domestic networks by the ISP. The present solution is compatible with any current domestic network as it implies no modifications to this standard network, although advanced configurations may be needed to make some changes to their parameters.

- **IoT Router**: An IoT Access Point with firewall capabilities. This router is wire-connected to the main router. It will only allow IoT devices to connect to it wirelessly. It will also implement extra features to provide security and privacy.

The proposal adds a whole new network, the one created by the IoT Router, to the existing one. In our practical scenario, this IoT Router is a Raspberry Pi Model 3b+, whose main specifications will be covered later. It implements Raspbian OS, which is a Debian based operative system, and it is working as the access point for all the IoT devices we configured as an example.

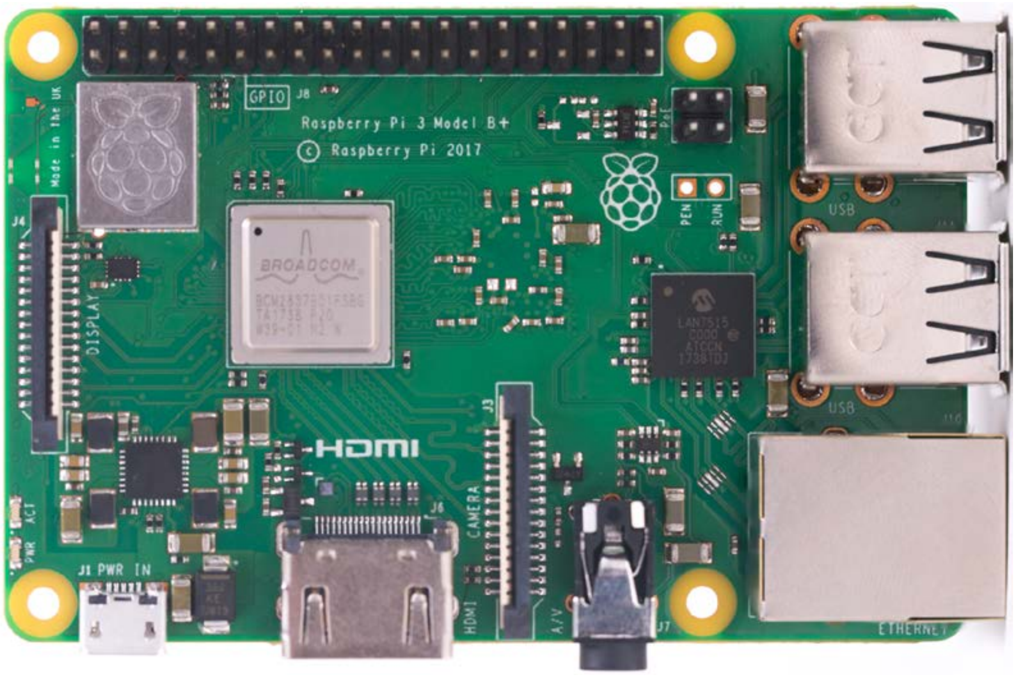

Figure 5 - Raspberry Pi Model 3b+ Board

It run several python scripts in order to make AP setup easier, user will only have to call the script within the command line of the system terminal.

For instance, if we type:

$> python wifi_setup.py <ssid> <psk>

It would configure a new AP whose SSID and PSK is determined from the data passed as first and second arguments. This AP will be visible by default, although it could be hidden as well, which basically means that it would not advertise itself to close devices, and therefore any user that may desire to connect to it would have to manually enter its SSID within the router's range. This can be done to increase the security as well as to discourage non-expert users from connecting normal devices to this AP. However, monitoring packets with a wireless card will end up showing traffic to this AP, and therefore revealing its existence and ubication.

The AP also implements an IoT multimedia software with a built-in firewall able to monitor the traffic of the devices in the network and allow/disallow certain types of packets or destination.

- **Monitor Mode**: will allow the user to check their status send specific commands to each IoT device (like turning off the lights, reboot, send now sensor information, etc.). This feature converts the router in an IoT Hub, where all the management is performed by users. This will also include the creation and maintenance of local databases in the AP that some devices might use to send any kind of information that may be gathered by their sensors.

- **Firewall**: write traffic rules based on different conditions such as device affected, packet type, addressee, etc. It will increase the security for all the IoT ecosystem, preventing devices for being tracked.

The configuration process that must be executed to have this scenario up and running consists in only 4 basic steps, which are the following:

- **Random Key Generation**: A new pair of <ssid, psk> is created in the Remote IoT Controller. It employs true random sources to generate a 32-byte key. The key is saved in clear in the EEPROM memory, so it is not erased when the device is shut down. Therefore, it is assumed that the remote is physically kept in a safe place away from intruders.

- **Secure Key Injection**: The controller sends the key to all the devices the user desires to initialize. It uses custom infrared protocols with a Diffie-Helman key exchange solution, implementing an elliptic curve with which a shared secret is generated to, in last instance, encrypt the data. This process is done once per device needed to be initialized.

- **IoT Router Initialization**: Some alternatives are provided on this step:
  - The first one is using a mobile QR reader to read a QR Code which encodes the SSID and PSK. User will have a readable string of characters representing this previously generated tuple. They will only have to log into the Raspberry Pi (IoT router) and manually execute the script, *python wifi_setup.py <ssid> <psk>,* typing the SSID and password, in order to have the Access Point running.
  - The second alternative is identical to the previous one but, instead of running the script in the device, it is done remotely by an SSH connection, executing the python automated script with the SSID and PSK values as arguments of the script.
  - The third alternative is using the Raspberry Camera module as a QR Code Scanner. It would work exactly the same way a QR is scanned with a mobile. However, in this case the camera module is not as good as the one included in most of the smartphones. Therefore, it is not currently possible to directly scan from the ESP8266 OLED screen. The solution is using the phone as the intermediate, scanning the OLED screen and representing again in a bigger format the QR code in its screen so it can be finally scanned by the RaspPi. All the details are covered in the software section.

- o Another alternative would be plugging the IoT Controller device to the USB port of the RaspPi and calling a script that is able to read the serial port information and extract the credentials from it. This script has been implemented as part of the project.
- o The last alternative would employ Infrared technology to transmit the information the same way it is done with IoT devices. Despite having designed the IR circuit, there were issues implementing a valid IR Library compatible with the one used in the controller (on an ESP8266). Therefore, this method was abandoned, although it is theoretically possible (but really complex) to program a library and make it compatible with the ESP8266 one.

- **Connection between IoT device and AP**: This step is automatically done once the AP is up and running and the device has received the Wi-Fi credentials. From this moment, all configured IoT devices will be actively and periodically looking for an access point whose SSID matches the one received in the transmission with the controller. Once they detect it, they will connect to it with the provided password.

At this point, the basic configuration is done, and the network should be running fine. From now on, it would enter into a maintenance and management phase, in which the user would be in charge for further manual configuration (like firewall rules or IoT device custom commands).

3.2. Hardware

The hardware used in this proposal consists in several devices and some circuits used as amplifiers for the infrared components. Many of the devices were purchased in a development board form-factor, as it significantly reduces the complexity of having to weld and design PCBs to put all the smaller parts together.

In a commercial scenario, it would be advisable to buy the components separately at a cheaper price and to design our own boards or PCBs to reduce the cost as much as possible.

3.2.1. ESP8266 NodeMCU as IoT Device

To simulate IoT device behavior, ESP8266 microcontroller embedded in NodeMCU[11b] development boards is used. These chips are low cost, and its main features are, for the NodeMCU devkit, the following:

- ESP8266 microprocessor, RISC 32bits CPU at 80MHz
- 32 kiB instruction RAM depending on the model
- 96 kiB user + system data RAM
- 4 MiB flash memory
- IEEE 802.11 b/g/n Wi-Fi
- 10 GPIO pins (soldered)
- USB tension regulated for powering

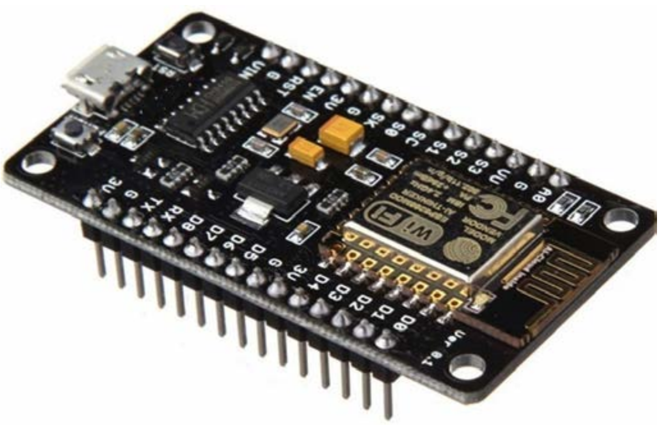

Figure 6 - ESP8266 NodeMCU Board

3.2.1.1. IR LED Transmitter Circuit Design

All IoT simulation devices have a circuit to amplify the power of the output IR LED. Despite already working if connected directly to the PINs, after testing, the range was found to be reduced and the Bit Error Ratio was significantly higher. Therefore, it was concluded to be worthy to invest a bit more to design an amplifier circuit to feed the LED with more current.

The designed transmission circuit is the following:

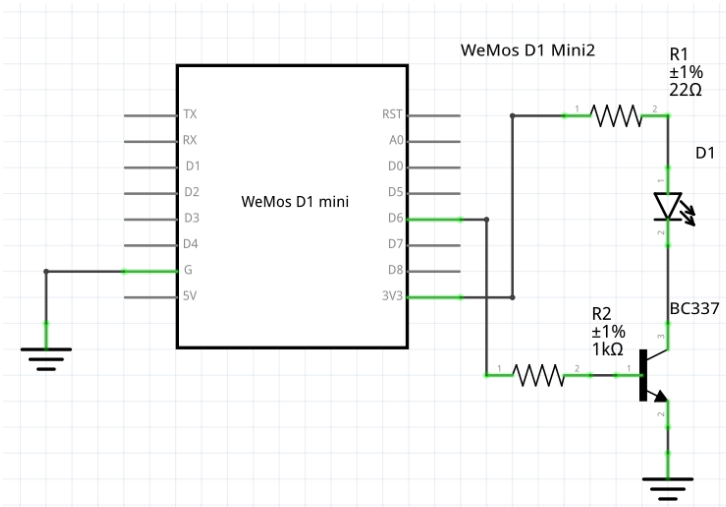

Figure 7 - IR LED Transmitter Circuit 1 LED

The current value in the LED (D1 in the diagram) was around 91mA in DC. The voltage between its terminals is 1.3 V. We have a security margin of almost 10mA, as the LED manufacturer recommends not to exceed 100mA of current and to have a voltage between 1.2 and 1.4V.

With this circuit, new testing proved the communication to be quicker and much more reliable. However, the range was still limited, and some packets got corrupted, increasing the transmission delay.

A redesign was made to the circuit above with the objective of fixing this, and a new circuit was proposed with 2 LEDs in series with a greater range. This was the design we finally chose for our final testing, as after analyzing the cost, we concluded the extra investment was worth it.

The final design transmission circuit for NodeMCU boards (IoT devices) is the following one:

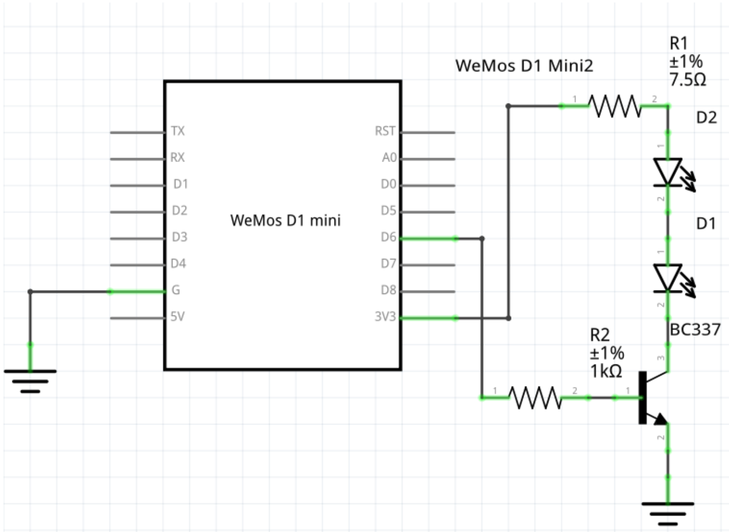

Figure 8 - IR LED Transmitter Circuit 2 LEDs

As it can be appreciated, the circuit is similar to the previous design, although there are now 2 LEDs in series and the values of the resistors were changed in order to assure a stable current of 93mA in DC and 1,3V in the diodes' terminals.

As we could not found a commercial value of 7.5Ω for the resistor. We substituted it by two resistors in series of 5.6Ω and 1.8Ω, which result in a total of 7.4Ω. This means that, in practice, the current in the diodes is slightly higher. However, this does not imply a redesign as it is still below the max current value of 100mA.

Also, the tolerances of 1% of the diagram resistors could not be found in the store. Therefore, the circuits implement resistors with 5% tolerance.

3.2.1.2. Bit Error Ratio Analysis 1 LED vs 2 LEDs

We made some calculations to theoretically determine distances and error probabilities in the transmission. The results are summarized in the following table:

ERROR PROBABILITY at 4 meters		
	1bit block	32bit block
1 LED	$6.4*10^{-4}$	$2*10^{-2}$
2 LEDs	$8*10^{-6}$	$2.6*10^{-4}$

Table 1 - Error Probability 1 LED vs 2 LEDs

As we can appreciate, the error probability was greatly reduced (around 100 times less probability of error) if using two transmission IR LEDs. As an extra IR LED cost is affordable, our design finally implements two LEDs in serial.

3.2.1.3. IR Receiver Circuit Design

The IR receiver circuit is simpler, as it works properly if connected directly to the GPIO Pins and it does not require any kind of adjustment.

The IR component used as the receiver is the TL1838[12b]. We chose this model due to its low price and correct performance. However, there are better but more expensive models in the market, such as the TSOP2238[13b], with greater performance at detecting false positives and filtering out infrared environmental interference.

The IR receiver circuit is the following:

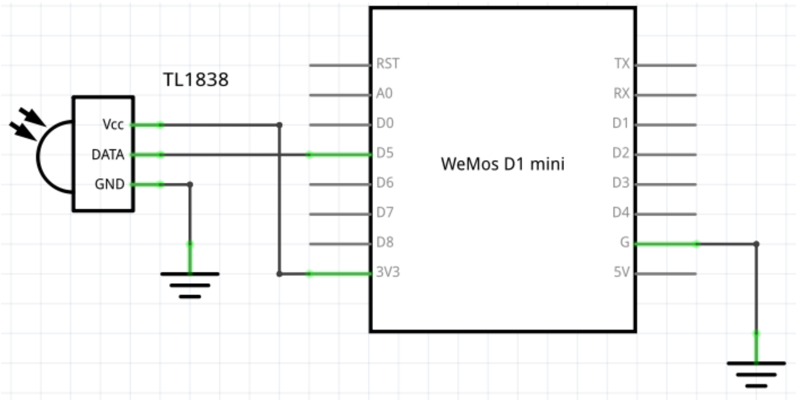

Figure 9 - IR Receiver Circuit

The IR receiver component has 3 terminals, Ground, Data, and Vcc. The design simply connects each one to their analog PIN in the NodeMCU board.

3.2.2. ESP8266 Heltec Wi-Fi Kit 8 as IoT Controller

Acting as the controller, we are employing a different board which implements the ESP8266 microchip as well. The controller is a crucial device is the design, as it is responsible for loading the key to each IoT Node by pressing a single button, as well as allowing the user to retrieve the generated tuple of SSID and PSK to configure the Access Point.

The features of the Wi-Fi Kit 8[14b] we are using are the following:

- ESP8266 microprocessor, RISC 32bits CPU at 160MHz
- 32 kiB instruction RAM depending on the model
- 96 kiB user + system data RAM
- 4 MiB flash memory
- IEEE 802.11 b/g/n Wi-Fi
- 17 GPIO pins (unsoldered)
- Onboard 128x32 dot matrix OLED display
- USB tension regulated for powering
- Onboard battery interface, with charge capabilities

Figure 10 - Heltec Wi-Fi Kit 8 Board

The key differences with the NodeMCU board are the OLED screen and the battery interface. Those are the main reason why we chose this model to act as a controller despite costing almost as twice as a NodeMCU board.

The OLED screen is used to display to the user some configuration menus (which are navigated via short press and confirmed the selection with a long press) as well as a QR Code. This QR Code is the way the device communicates with the user to allow him to retrieve the randomly generated key. This QR Code can be easily scanned using a mobile phone camera.

The battery interface would permit freedom of movement to the user, allowing him to freely go around his home aiming at his new IoT devices to communicate with them and send them the AP key. This battery can also be charged by just plugging in the board with an USB cable to a computer.

As this board is completely different, it also outputs a different current level in their GPIO Pins, which turns out to be higher than in a NodeMCU board. Therefore, in this case we decided to just connect the IR LED and Receiver directly to the board, as the current supplied to the LED was enough and the communication worked correctly this way.

3.2.3. Raspberry Pi Model 3b+ as the IoT Hub

The present design proposes creating an isolated network for the IoT devices inside the existing domestic one. Therefore, a device capable of providing Wi-Fi connectivity, router and firewall features, as well as scripting support was needed. We chose a RaspPi due to its versatility and upgradeability, as well as the wide variety of open source libraries that can be used to provided extended functionalities.

The Raspberry Pi 3b+ has the following features[15b]:

- 64bits 1.4GHz CPU
- 1Gb LPDDR2 SDRAM
- Wi-Fi module at 2,4 and 5GH, IEEE 802.11b/g/n
- Bluetooth 4.2 Module
- Ethernet over a USB 2.0 port
- Full size HDMI Port
- Camera support Port
- Touch Screen support Port
- Video and Audio in Stereo Ports
- Micro SD Card Port to store the Operative System and Data
- 40 GPIO Pins

Figure 11 - Raspberry Pi Model 3b+ Ports and PINs

In order to load the generated key to the RaspPi and create an Access Point, three alternatives are provided:

- Connecting to it via an SSH session and running the already programmed python script (which will be described in detail in the Software section), manually inserting the credentials information.

- Scanning the QR code with the RaspPi Camera Module. In this case, the camera focus quality is not good enough to read it directly from the Heltec Wi-Fi Kit 8 screen. User would need to read it with the mobile phone and a suitable application capable of switching between decoded information and read QR Code itself. This way, the

RaspPi camera would finally be able to read a bigger code from the smartphone screen.

- Injecting the Code using IR communication between the RaspPi and the ESP8266 Controller. However, this alternative was discarded due to both software (which are covered later in the Software section) and hardware issues by which the implementation of this alternative could not developed.

3.3. Software

The software part is the one that took most of the time to complete. Its core part is the design of a communication protocol to communicate securely and as fast as possible between pair of devices (a controller with an IoT device). As this communication requires the use of interruptions whenever a signal is detected, it is not trivial to make a design beforehand. Therefore, the initial design was iterated many times (sometimes adding new functionalities, other times just implementing the solution differently due to poor performance) until obtaining a code that performed efficiently.

However, many other software solutions were adopted to cover other requirements of our design. In the case of everything concerning the ESP8266, C programming language was used; while in the case of the RaspPi, we opted for different versions of Python. The reasons for choosing different languages in each platform are covered later.

There are many other features implemented apart from the custom IR communication protocol itself. In this section all of them will be covered in detail, explaining how everything works and justifying the implementation decisions that were taken in each case.

Finally, in the Annex part a full documentation of every program and script will be provided, as well as the full source code for consultation.

3.3.1. ESP8266

The ESP8266 was used for both controller and IoT node simulation. Therefore, it was required to develop two different programs in C to allow them to communicate with each other and to satisfy the development requirements. Two programs were coded:

- **Receiver Program**: It is implemented in every IoT Node. It waits for a communication from a controller, interacts with him, and saves the received credentials in the EEPROM memory. Finally, it connects to the Access Point corresponding to the credentials received.
- **Controller Program**: It runs on the ESP8266 Wi-Fi Kit 8 board. Beside implementing the same communication architecture and routines than his counterpart, it also implements other functionalities, such as true random SSID and key generation, OLED screen management, QR Code encoding and display and user interface menu. Unlike every receiver, the Controller will not try to connect with the Access Point.

Every one of these features will be broken down and explained in depth in the following sections, including all technical information.

Due to the high level of detail, the following subsections are not included in the general index of the document. The structure of the ESP8266 part is the following one:

- ESP8266 Controller
    - OLED Screen Management and User Interaction
    - EEPROM Memory Management
    - SSID and PSK Key Random Generation
- ESP8266 Receiver
    - EEPROM Memory Management
    - Connection to AP
- Controller and Receiver Bidirectional Transmission
    - IR Communication Design Process
    - Type of Packets and IR Packet Frame Design
    - Data Encryption
    - IR Protocol Communication Diagrams
    - Time Evolution Analysis

3.3.1.1. ESP8266 Controller (Heltec Wi-Fi Kit 8 Board)

In this section the major details of the code programmed for the Controller device are discussed and analysed in depth. Nonetheless, attached to this report there is a HTML documentation of the program generated with Doxygen, and also the full code is available for consulting in the **Annex I**.

OLED Screen Management and User Interaction

As showed in an image in the hardware sections, the board used as a controller has a built-in OLED display, with dimensions 128x32 pixels. The library *Adafruit_SSD1306*[16b] is used to be able to print information in the screen. Once it is initialized, we can use Adafruit library to print information on the OLED screen the same way we would print it on the serial console.

User interaction with the screen is limited to two selection menus (mainly used for the development, they might be suppressed in a commercial version) and a single button. There are two menus that prompt for user input when the device is turned on:

- **Node Selection Menu**: Allows the developer to force communication with an already saved device, in case the credentials were changed. The default option is adding a new device by using a broadcast packet, however, an already initialized device will ignore this kind of packets.
- **Encryption Selection Menu**: Allows the user to communicate in an encrypted way (default option) or sending the credentials in clear. Encrypting increases the security but also the time needed to send the data. Probably, a commercial device should omit this menu and force the device to perform every communication using encryption.

User input is done via the built-in flash button in the board. A single press would switch among options and a long press would confirm the selected one. Screen displays the highlighted option in every moment. Once user confirms both menus, the controller tries to communicate with a node.

Once communication is successfully completed, the screen will display the QR code which encodes the credentials generated and transmitted. In order to generate this pixel-based code a library called *QRCode*[17b] was used.

Although the library was conceived to print a QR code on a terminal, it was trivial to print it on the OLED screen with the screen library's printPrixel() function. The amount of information that can be encoded on a QR Code depends on both its size and its Error Correction (EC). The higher the size and the lower the error correction capabilities, the larger the amount of data that can be encoded.

The QR codes are classified according to their version and EC. The size of it is given by the formula: size = 4*version+17. As our screen is limited by its height of 32 pixels, the only versions we can implement are 1 (21x21), 2 (25x25) and 3 (29x29). Version 4 would surpass the height limit, as it would have a size of 33x33.

Also, it is important to check the amount of information that is planned to be encoded. In our case, we are encoding a SSID (16 bytes) and a PSK (32 bytes). That makes a total of **48 required bytes**. Also, it would be advisable to add some extra characters to split both strings, so a user can distinguish the SSID and the PSK after reading the code.

Due to these limitations, there is only one possible combination which meets all the requirements. That one is version 3, 29x29, with a Low level of Error Correction. This code will allow us to encode up to 53 bytes, which means we have 5 spare bytes to add some special characters (such as spaces, line breaks…) that will separate both strings.

A comparative table of the different versions and their impact in the QR Code final capacity are summarized in the following table:

Version	Size	Error Correction	Mode			Version	Size	Error Correction	Mode		
			Numeric	Alphanumeric	Byte				Numeric	Alphanumeric	Byte
1	21x21	LOW	41	25	**17**	7	45x45	LOW	370	224	**154**
		MEDIUM	34	20	**14**			MEDIUM	293	178	**122**
		QUARTILE	27	16	**11**			QUARTILE	207	125	**86**
		HIGH	17	10	**7**			HIGH	154	93	**64**
2	25x25	LOW	77	47	**32**	8	49x49	LOW	461	279	**192**
		MEDIUM	63	38	**26**			MEDIUM	365	221	**152**
		QUARTILE	48	29	**20**			QUARTILE	259	157	**108**
		HIGH	34	20	**14**			HIGH	202	122	**84**
3	29x29	LOW	127	77	**53**	9	53x53	LOW	552	335	**230**
		MEDIUM	101	61	**42**			MEDIUM	432	262	**180**
		QUARTILE	77	47	**32**			QUARTILE	312	189	**130**
		HIGH	58	35	**24**			HIGH	235	143	**98**
4	33x33	LOW	187	114	**78**	10	57x57	LOW	652	395	**271**
		MEDIUM	149	90	**62**			MEDIUM	513	311	**213**
		QUARTILE	111	67	**46**			QUARTILE	364	221	**151**
		HIGH	82	50	**34**			HIGH	288	174	**119**
5	37x37	LOW	255	154	**106**	11	61x61	LOW	772	468	**321**
		MEDIUM	202	122	**84**			MEDIUM	604	366	**251**
		QUARTILE	144	87	**60**			QUARTILE	427	259	**177**
		HIGH	106	64	**44**			HIGH	331	200	**137**
6	41x41	LOW	322	195	**134**	12	65x65	LOW	883	535	**367**
		MEDIUM	255	154	**106**			MEDIUM	691	419	**287**
		QUARTILE	178	108	**74**			QUARTILE	489	296	**203**
		HIGH	139	84	**58**			HIGH	374	227	**155**

Table 2 - QR Code Version, Size and Capacity Comparison

As we can see, in the implementation adopted there is low error correction, which may cause a single dead pixel in the OLED screen to make the whole QR Code unreadable. Also, this limitation prevents the design from using a larger private key of, for instance, 64 bytes, as it would not be possible to be encoded in a 29x29 code.

Another critical issue of using such a small screen is the difficulty to read it with a normal camera. The size of the pixels and the screen backlight can prevent some mobile phones from detecting and scanning the code.

After many attempts, we found out that inverting the QR code colours would make it easier for a phone to read it. Instead of using white pixels over a black background, we turned the whole screen white and then turned off the QR turned on dots. This fix works as it helps the camera to adjust its light exposure and focus the code with more precision.

A good idea would be using a NodeMCU board as the IoT Controller and connect to it an external 128x64 OLED screen. This would make things easier. In this case, we could:

- a) Keep QR Code version and increasing every QR dot to 2x2 pixel size, making it easier to scan, or
- b) Keep 1 pixel per QR dot and use QR version 11 (61x61) with Error Correction High. This would allow us to increase the size of the PSK key to 64 bytes and still being able to encode it, at the same time we are enhancing the robustness of the QR Code itself.

However, a larger screen would make the controller less comfortable to use, reducing its portability and its battery life.

EEPROM Memory Management

In order to make information persistent, the built-in EEPROM memory that the ESP8266 integrates is used. Although there is more memory available, the controller only uses 256 bytes of memory.

There is a library (included in the ESP8266 own devkit) called *EEPROM* which makes writing/reading easier. However, some custom functions are included on top of this library in order to read/write information in a much simpler way.

Each memory address can store up to one byte of data. The memory mapping, which gives information about the where each piece of data is stored in the memory, is the following:

Controller EEPROM Mapping (256 B)		
Address	**Information**	**Total Size**
0x00 - 0x0F	SSID	16
0x10 - 0x2F	PSK	32
0x30	Number Nodes	1
0x31	Node 1	1
0x32	Node 2	1
----	----	----
0xFE	Node 206	1
0xFF	Control Char	1

Table 3 - Controller EEPROM Memory Mapping

The first 48 Bytes are reserved for the SSID a PSK information of the Access Point.

The very next Byte, the 0x30, indicates the number of nodes that have been initialized and have the credentials in their memory. This value allows the device to know where to stop reading the memory for Node IDs.

As the Number of Nodes is only one byte, it restricts the number of nodes to 2^8 = 256 devices. Nonetheless, as some of the addresses are used for the Wi-Fi credentials, the number of storable devices is lower: 206. However, this number should be more than enough for most of the cases.

Finally, there is an important char located in the last used memory address, the 0xFF. The Control Char address is the very first one that is read when the device is booted up. It will allow it to realize whether the device has been initialized in the past or not.

If the value read is equal to the character 'Y', or 0x59 in hexadecimal, it means the memory has already data stored in the other addresses. Otherwise, the device will randomly generate a SSID and PSK and initialize the number of nodes to 0.

SSID and PSK Key Random Generation

One of the most important features of the ESP8266 is its ability to generate true random number from high level entropy sources. However, the stock random generation function in C is not truly random, so an external library had to be used. This library is the ESP8266TrueRandom. This library gets the entropy by reading the ESP8266 internal hardware random number generator register or by measuring the voltage level of the A0/TOUT pin.

- **SSID Generation**: The SSID has 16 bytes, with the format: *IOT_aaaaaa-nnnnn*. It starts with a prefix "IOT_", 6 characters, a separator '-', and 5 numbers.
  Each 'a' character represents any letter, lower or capital, of the English alphabet (52 possibilities per character). The 'n' character can be any number from 0-9 (10 possibilities per character). The randomness of the SSID is not critical, as it will end up being a public value when announced by the Access Point.
- **PSK Generation**: The method used for generating the PSK is quite simple.
  - The random() function of the custom library, which generates a number of 32 bits, is called 4 times in a row.
  - Afterwards, each number is appended resulting in a 128-bit random number.
  - Finally, we split the 128-bit chain in 32 chunks of 4 bits, resulting 32 random hexadecimal numbers.
  - These hexadecimals are converted from 4-bit HEX Char to their ASCII representation (8 bits). This conversion duplicates the length of the key up to 256 bits, while still assuring the readability of the information. For instance, character 'C' in Hexadecimal would be 0b1101, while its conversion to ASCII is 0b1000 0011.

In the end, despite having 32 chars or 256 bits of information, the real entropy comes from 128 bits due to the padding.

3.3.1.2. ESP8266 Receiver (NodeMCU Board)

Alike the Controller part, this section covers the code designed specifically for the Receiver devices. The documentation of the full program is attached to this report in HTML, and the code can also be found in the Annex II of this report.

EEPROM Memory Management

Analogously to the Controller, the memory mapping in the receiver uses the same principles and mechanics to read and write persistent data. In this case, the total memory used is 64 bytes. In these devices, the amount of stored information is lower as there is no need to store the whole list of initialized devices.

It also uses the same custom functions to read/write with more easiness to the EEPROM. The way the information is organized inside a receiver's memory is the following:

Receiver EEPROM Mapping (64 B)		
Address	**Information**	**Total Size**
0x00 - 0x0F	SSID	16
0x10 - 0x2F	PSK	32
0x30	Dev ID	1
0x31 - 0x3E	Unused	14
0x3F	Control Char	1

Table 4 - Receiver EEPROM Memory Mapping

Out of the 64 bytes assigned, 14 are not used. However, I considered adequate to allocate an amount of memory resulting from a base power of 2 (2^8).

Also, as it happens in the controller, there is a control char that, if equal to the character 'Y' means that memory was initialized. In this case, the device would directly read the SSID and PSK and try to connect to the Access Point.

Finally, unlike the case of the controller, where there is no need to store its own device ID as it is hardcoded and it is always equal to 1, the receiver needs to store his ID in case of power shutdown. The ID is used for identifying the device in both infrared communication with the controller and to define his static IP, as we will detail in the following section.

Connection to AP

The ESP8266 Arduino devkit includes a library to handle the Wi-Fi connectivity in the device. It also includes a lightweight IP protocol stack in the chip. The library is called *WiFiClient*[18b], and it allow the device to act as an Access Point, or as a station and connect it to another AP.

As it will be explained later, in the Access Point section, the subnet used for the IoT ecosystem is the 192.169.100.0/24. The gateway is the 129.168.100.1, and each IoT device asks for a static connection based on his ID. Each device will request a connection with an address of the type 192.168.100.*devID*.

While connected to the AP, the device will keep listening for infrared communication from a controller sending his own ID. This will allow a controller to change the device's stored credentials.

As the application layer definition is not object of study of this work, once the device is connected to the AP its functionality will be limited. However, it will still answer to some lower layer protocols such as the ICMP, and the device will answer correctly any ping request.

3.3.1.3. Controller and Receiver Bidirectional Transmission

The act of transmitting the information using infrared can be considered the core functionality of this project. Infrared was not designed for a reliable bidirectional transmission of information. Therefore, it was needed to develop from scratch new functions that provided these requirements.

The creation of this part of the project was also the most time consuming one. It involves asynchronous calls as well as constant interaction with the hardware (IR receivers and senders) that make the behavior kind of unpredictable. In the end, a lot of time was invested in developing, testing and iterating each new design until the final version, which can be considered completely functional, reliable and secure.

Due to its nature, the initial design that was made to code all the logic needed continuously evolved during the implementation process. Given the lack of experience and knowledge in IR communication in the beginning of this process, many errors were made and important things to consider were not.

Therefore, this significantly delayed the predicted development schedule, as a lot of iterations were finally needed. Moreover, not only the software suffered changes, but also the hardware. At a certain point, there were some erratic behaviors that were not seemed to be caused by the software. We suspected that the transmitting LED was not receiving enough current, ending up redesigning its circuit and including two LEDs fed with more power.

The actual working version has been well optimized; however, it is not perfect. There are many small details that could be implemented to achieve further time improvements and memory reduction. However, due to personal schedule restrictions and the fact of already spending almost 4 times more time than intended for this part, I decided to leave it as it is in order to have some time left for the other parts of the project.

In each of the following subsections, all remarkable features of the bidirectional communication will be explained in detail. Nonetheless, I will try to avoid long code writing to keep the document clean and readable. In case a feature produces any confusion in the reader, please refer to the full source code that will be annexed at the end of this report. Also, the documentation may be useful to retrieve further information which may clarify a doubt.

IR Communication Design Process

Before diving into details, it is important to take a glimpse at the iterative process of the design, from the very first working version to the current one:

1. **Unidirectional testing**, from controller to receiver and vice versa. Sending of some test data to see behavior.
2. **Bidirectional testing**. Controller starts communication sending a packet representing a number, receiver sends back same number plus one, and so on.
3. **IR Library Protocols testing**. Analysis of every protocol specifications and behavior in practice, until choosing the best one: **RC6**.
4. **Definition** of each **type of packet** needed.
5. **Design of the frames**. Each frame contains not only data as the payload, but also other useful information such as the destination, or the type of packet.
6. **Design of the communication phases:** Handshaking, data sending, and connection close.
7. **First working version**. Information was sent using a Stop&Go protocol, without any error correction mechanism. It took around 8 seconds to transmit without encryption.

   Each packet was sent periodically until receiving its ACK and switching to the next one. Drawbacks: Time inefficient, error if there were two consecutive packets with identical payload (as there was no space for a packet counter in a data frame).
8. **Handshaking optimization**: in the case of concurrence (when 2 devices are listening at the same time for the controller), the initial handshaking method caused devices to collide with each other when trying to answer simultaneously a HELLO packet.

   The solution was simple: the HELLO packet sequence was extended, including a random number challenge, so only the receiver which generated that same random number can continue the communication with the controller.
9. **Protocol and Time optimizations.** The library was slightly modified to reduce the time it takes to send a packet. Also, the RC6 protocol uses a frequency of 36Khz. To avoid signal losses due to frequency mismatch, it was changed to match the frequency used by our IR receptor: 38Khz.
10. **Second working version**. The Stop&Go protocol was substituted by two burst sequences (for both SSID and PSK) with an ending ACK each. Also, a checksum for error detection was included in the last packet. This version solved many issues of the first one, as well as some hardware erratic behavior. Also, it reduced the time needed to send all information. It took around 5.5 seconds to transmit all information in clear.
11. **Design of the encryption**. A new communication phase was included to provide support for the public key sharing using a Diffie-Hellman Key Exchange method.
12. **Further optimizations**. The key exchange significantly increased the total transmission time. Some adjustments and minor changes were made to reduce it.

13. **Third working version**. Added some failure conditions to prevent endless loops in case of communication interruption, as well as more optimizations. It reduced the transmission delay: 2.2 seconds in clear, 4.5 seconds encrypted.

14. **Fourth working and current version**. This version reduces the transmission time around a 15% compared with the previous one. It reduces the number of total transmitted bursts from 3 to 2, as it includes the controller's public key in the data burst it sends. The resulting time delays are 2.0 seconds in clear, 3.7 seconds if encrypted.

In general, each new version has been adapted to the circumstances and requirements of the current problem. This means that the transmission protocol has evolved from a very generical one able to transmit any kind of information but in an inefficient way to the current one, in which we are transmitting three times more information in three times less time, at the cost of versatility and adaptability. The code has been optimized for this particular case. However, this does not necessarily mean any additional data can no longer be transmitter, but the complexity of allowing it would translate into an additional cost in terms of effort when programming and optimizing it to fit the rest of the code.

Despite the loss in versatility, the use of infrared communication is not a widely adopted solution. The decision was done to meet a security requirement. Therefore, it is very unlikely that the transmission functionality that has been developed requires any remarkable upgrade in the future.

Type of Packets and IR Packet Frame Design

In order to make the communication possible, there was a strong need to design differentiated packets that allowed the receiver to know beforehand the type of content that goes inside a frame.

Therefore, the definition of the type of packets was done at the very beginning of the design of the protocol. However, as the program evolved, so did the initial definition. Some new types were added, and some others were removed to meet each new specification.

The final version implements two different sizes for the frames: 16 and 36 bits.

16-bit packets

Used in all ACK frames. This was done to reduce the transmission time for the acknowledge and, therefore, improve the overall efficiency of the protocol. The structure of this frame is the following:

ACK Frame Structure (16 bits)															
Packet Type			**Is From Controller**	**Payload** (12 bits)											
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Table 5 - ACK Frame Structure 16 bits

The 3 most significant bits are used to identify the packet. In this special case, as the ACK is the only packet which has 16 bits length, it could be suppressed. However, it is interesting to provide an identifier in case a future improvement includes another different 16 bits length packet. The 3-bit identifier corresponding to an ACK is 0b100.

The bit 12 is a flag used to differentiate packets coming from controller (if set to 1) and packets coming from IoT nodes (if set to 0). This bit is not strictly necessary, however, it was included as an additional integrity measure, as we wanted to leave the payload with an even number.

The 12 less significant bits are reserved for the ACK payload. ACKs are sent after receiving a final data packet identifying the end of a burst of information packets. The payload is filled with the information of the number of received packets. Again, it is not strictly needed as we are including a much trusty checksum method for integrity verification, but it provides an extra layer of reliability.

36-bit packets

These packets are used for all the other stages of the communication. The type of packets is limited to 8, as we are using a 3-bit field as their identifier in-frame. The 36-bit possible packet types are the following:

- **Hello New**: 0b010. This type of packet is responsible for the communication establishment with a new node. It is always broadcasted from a controller to all listening devices, including in its payload the new nodeID which the answering node will adopt. It is answered by an IoT node with a packet of the same type, including in the payload a random sequence as a concurrency prevention mechanism.

- **Hello**: 0b001. This packet is sent after a Hello New, in case a new node is contacted. It can also be the first packet of the communication in case we are connecting to a known node. It is first sent by the controller, with the payload equal to the selected random sequence received by the node in case there was a Hello New; or filled with the existing node ID, in case an existing node is contacted. The answer is always another Hello packet with the nodeID in the payload.
- **Hello Encrypt**: 0b000. It has the same functionality than a normal Hello packet. However, as there is not room for an extra bit indicating an encryption flag, a new packet type was set for the controller to indicate a node that information will be sent encrypted, and therefore it should prepare in first place for a public key exchange procedure.
- **Data**: 0b011. The type of packet whose payload is actual information. It will come inside a burst of variable number of Data packets.
- **Data End**: 0b101. It indicates the ending of a burst of Data packets. Its payload also contains information. However, since the inclusion of the checksum, the 32 bits of the payload in the Data End packet are always the checksum corresponding to the payloads of the entire burst of packets.
- **Data End Encrypt**: 0b110. It also indicates the end of a burst, as well as specifying that the information was sent encrypted and it is needed to decrypt it before being able to read it.

The role of each packet type will be explained in more detail when reviewing the actual communication workflow of the infrared protocol.

The structure of a 36-bit frame varies depending on the type. This was done to optimize the payload in the data frames, where the destination ID information was removed to add an extra octet of payload.

The frame structure of any variation of a Hello packet is the following one:

HELLO Frame Structure (36 bits)																	
Packet Type (3bits)			**destID** (8bits)								**Is From Controller**	**Payload** (24bits) …					
35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18
… **Payload** (24bits)																	
17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Table 6 - HELLO Frame Structure 36 bits

In comparison with the ACK frame, the extra 20 bits give more room for extending the payload by 12 bits (from 12 to 24) and including an octet for identifying the addressee of the packet.

However, once the handshaking sequence is done, it is guaranteed we are talking with the right node as all the others will remain silent until the next Hello sequence. Therefore, there is no need to keep including a destination ID header anymore. By assigning this space to the payload, we are increasing the packet information efficiency by a 50% (from 24 to 32 bits).

The frame of every Data packet has the following structure:

DATA Frame Structure (36 bits)																	
Packet Type (3bits)			**Is From Controller**	**Payload** (32bits) …													
35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18
… **Payload** (32bits)																	
17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Table 7 - DATA Frame Structure 36 bits

This implementation of two different structures for a same packet length is possible since the identifier of a packet, the packet type field, remains in the same position if both cases. In the program logic, after receiving a frame, the first thing the code does is looking for the type of packet to determine its structure.

This design decision was taken in an attempt to optimize the protocol. Allowing more useful information in a packet means fewer packets to transmit, reducing in the end the global transmission time.

Data Encryption

The ultimate objective of this research is to provide the user a safe alternative to load Wi-Fi credentials into their IoT devices without compromising easiness of use. The use of infrared communication to achieve this objective constitutes, on its own, a much secure process than using other technologies, as WPS. However, this fact is not a reason to avoid including additional security measures.

With the inclusion of a Diffie-Hellman based encryption scheme we are increasing the protection strength to a point in which it can be assured that data will never be compromised by intercepting the communication in any case. At this point, it would be much more practical to simply raid the house and steal the information or devices physically (and this circumstance is far beyond our study).

Both parts in the communication implement in their code the same library to generate the keys that will be ultimately used for encryption and decryption. This library is *curve25519-donna*[19b].

This curve is widely used in cryptography due to its 128 bits of security and calculation speed when deriving new keys. Also, it is free to use, as it has not been covered by a patent. The library makes use of this key for generating public keys from the private ones.

In the code, the full process, by which the shared key is finally generated by combining both public keys from controller and node, which, at the same time, are derived from different private keys, is the following:

1. **Private Key Generation**

   A 32-byte key is randomly generated. This is done outside the curve25519 library. In our project, we use the same library used for the random generation of the Access Point credentials.

2. **Public Key Generation**

   A 32-byte public key is calculated from the private key and a basepoint which is usually a constant (in the code, uint8_t basepoint[32] = {9}]. After having created both private key and basepoint, using the library function curve25519_donna() will return a public key calculated using this elliptic curve.

3. **Public Keys Exchange**

   After both devices have generated their pair of public/private keys, they send the public key to each other. This is done using the designed infrared communication protocol described in this paper. As we are using infrareds, the probability of being intercepted in unlikely to happen. Even if it is sniffed, the Diffie-Hellman scheme prevents the attacker from decrypting the communication as he would also need the private key that is never shared.

4. **Shared Key Generation**

   Now that each part has the public key of the other, they can generate the shared key, which is the one that is finally used for the encryption/decryption of the information. With the same library function, curve255519_donna() called with the arguments my *Private Key* and *his Public Key*, the 32-byte Shared Key is generated.

Both keys should always be equal due to the mathematics underlying the key generation.

5. Data Encryption

The encryption is only limited in our case to the Access Point private key of 32 bytes. There are multiple ways of encrypting the data, as well as different available libraries based on initialization vectors and nonces to avoid encryption key leaking due to the overuse of the key. In the project none of these libraries is used, due to the particularities of our case. As it can be appreciated, the length of the data to encrypt matches with the length of the shared key: both are 32 bytes long. Therefore, a simple bit-level XOR (eXclusive OR) function is used to encrypt and to decrypt the information:

- Data Encrypted = Data Clear XOR Shared Key
- Data Clear = Data Encrypted XOR Shared Key

The XOR logic table is very simple, and it can be resumed in the following table:

XOR Logic Table		
INPUT		OUTPUT
A	**B**	**A XOR B**
0	0	0
0	1	1
1	0	1
1	1	0

Table 8 - XOR Logic Table for Encryption

As we can see, if we apply the XOR operation to any given value against itself, the result is always 0. For example, [x XOR A] XOR A is always equal to x as: 0 XOR 0 == 1 and 1 XOR 1 = 0. This make XOR operation a very simple and fast manner to encrypt and decrypt information in the case the information to encrypt matches the encryption key in length.

It is important to remark that each new communication implies the repetition of this process. Therefore, a new private key will be generated. Also, as the shared key is only used once to encrypt the information, there is no need to use nonces (random numbers of a single use that modify the key in each use) or initialization vectors and the encryption complexity is significantly reduced. If the key length was higher, for example 64 bytes, using the same key to encrypt two blocks of 32 bytes would reduce the scheme security. In fact, the more the key is reused, the weaker it becomes.

IR Protocol Communication Diagrams

The code implementation of each communication stage and process described above could be resumed in the following simplified workflow:

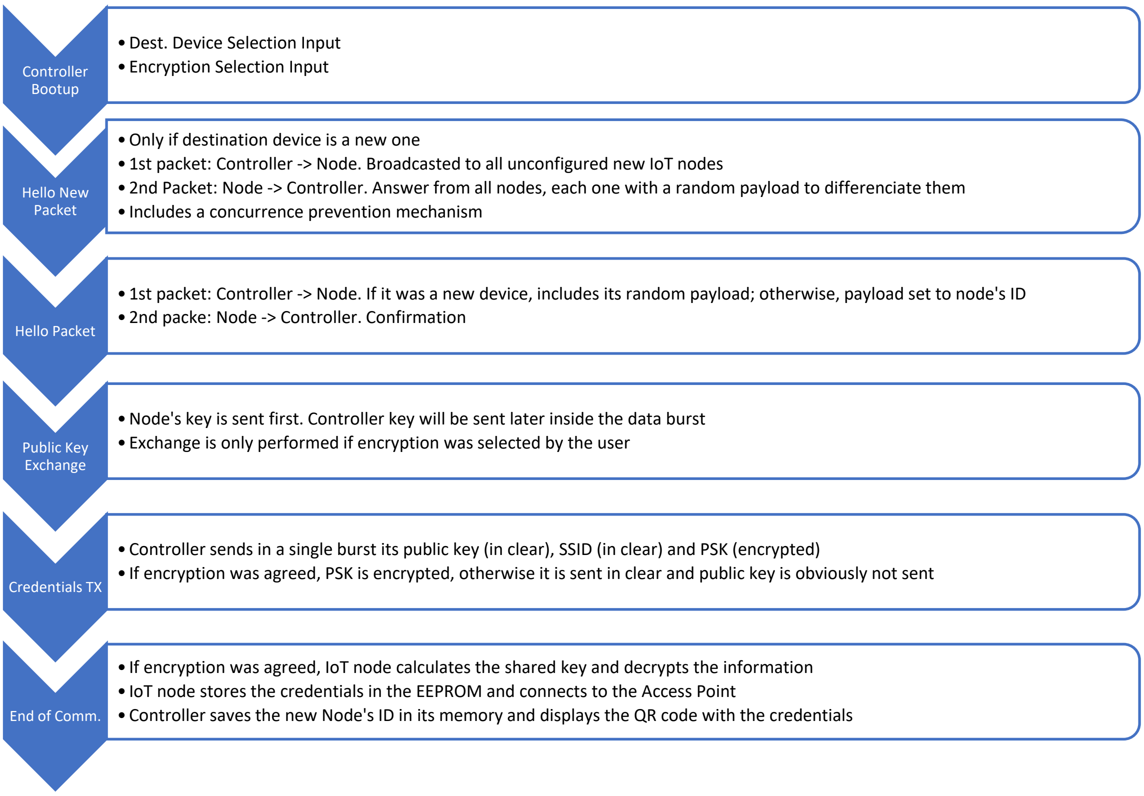

Figure 12 - IR Protocol Transmission Summary Workflow

However, this workflow does not reflect the complexity of the communication specific phase itself. As a detailed alternative, the following graphic focus on a much lower level, exposing the way each packet is sent in each stage. It represents the most complex case: connection to a new node, using encryption.

Also, it gives a clear answer to how the overall transmission sequence is made, how and when a packet is sent, as well as the time spent in each one of the communication phases. The details of the timed packet diagram are discussed below its figure.

In the following pages two time-diagrams corresponding to the two most recent versions are represented and explained in detail. As new versions have been programmed with further optimizations and upgrades, it is interesting to contrast two different approaches to see the difference and understand how each development decision may impact the project in terms of performance, usability, adaptability and memory.

3rd Working Version Transmission Diagram

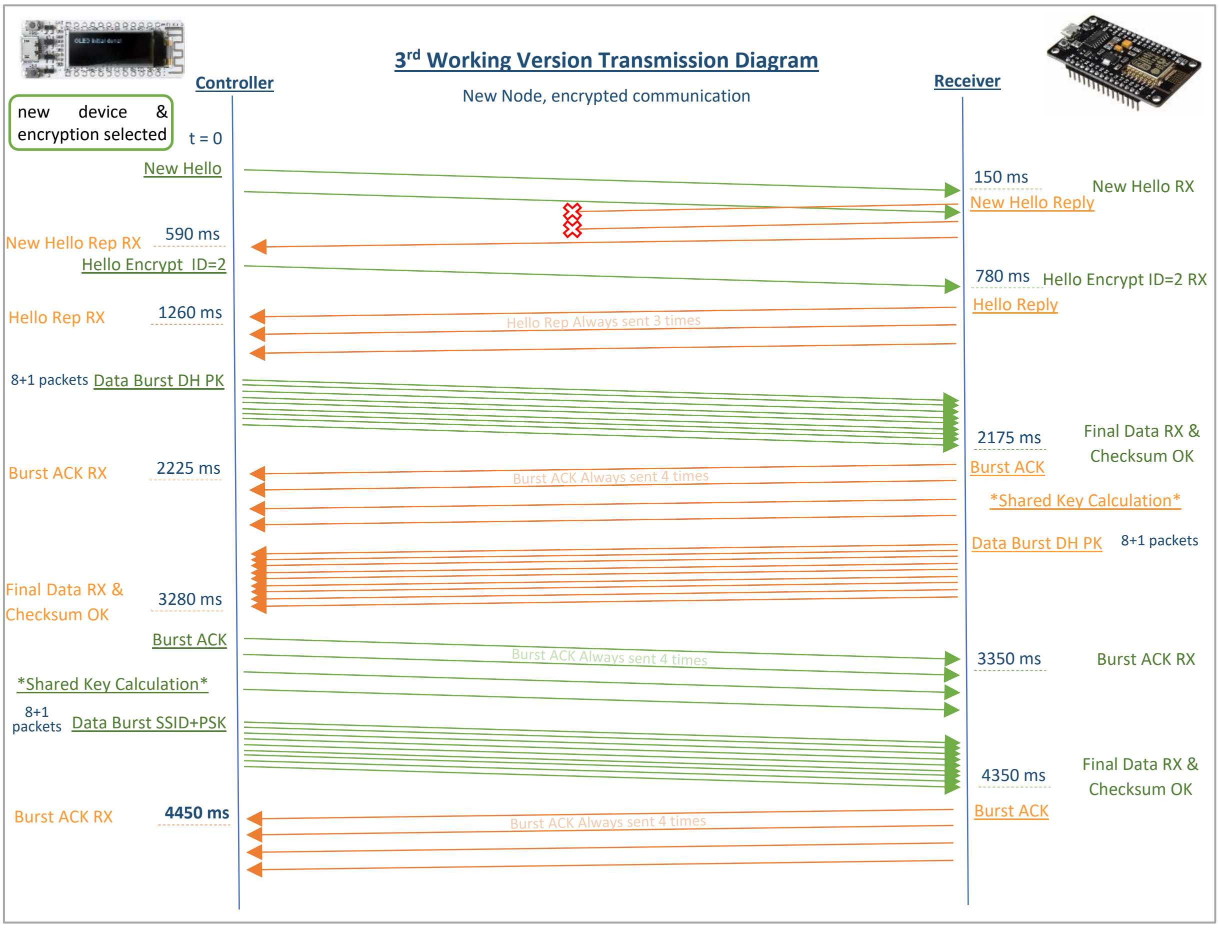

Figure 13 - 3rd Working Version Transmission Detailed Diagram

The above diagram has been extracted directly from the communication terminal logs in a real scenario communication between our controller and receiver, **using the third working version**. Every single transmitted packet has been represented, and the reflected times (in milliseconds) are exactly the ones that took each of the communication phases to finish, based on the logs.

It starts with the transmission of the first New Hello broadcasted packet once the controller device is switched on and the new device and encryption menus are confirmed. In this moment the controller will periodically transmit a packet until receiving a response. In our scenario, the first packet is usually the one received and replied; however, the response will not reach the controller before it sends a second New Hello.

The New Hello initially sent by the controller will have the assignable address ID in its payload (in this case, it was an id=2). Therefore, the receiver knows that the next hello packet, if containing a 2, will be sent towards it. The New Hello Reply implements a concurrence prevention mechanism by which the payload will be filled with a random number of 24 bits.

This New Hello Reply is also periodically sent until stopping receiving New Hello packets. This moment is the most critical one in the implementation, as the device are not synchronized yet and collisions may happen. As a countermeasure, this New Hello Reply transmission period has been randomized, to avoid multiple collisions due to a permanent synchronization that may infinitely delay the handshaking stage.

In the diagram, the initial 2 New Hello Reply packets where lost probably due to the collision with the second New Hello packet transmitted by the controller. However, finally the third packet made it to its destination informing the controller that a new unconfigured node is listening.

The controller will stop transmitting New Hello and immediately transmit a Hello. Collisions may still happen, as receiver keeps periodically sending New Hello Reply packets. Sending timers were optimized to prevent collisions at this stage, which translates into the first or second Hello packet is usually received by the receiver.

In this test, the first Hello packet was correctly received in the IoT node. In this moment the node will send three times in a row a final Hello Reply packet. Although the first one is correctly received, controller will keep the reception window opened, as it is aware that the receiver will send it three times and, until finishing, it will not be able to listen to any incoming packets.

Due to the initial configuration input by the user, the encryption mode was selected in the above diagram example. Therefore, the Hello packets sent by the controller are, in reality, Hello Encrypt packets, which tells the receiver node that communication will be done encrypted, and they first need to exchange their public cryptographic keys.

Once the Hello is acknowledged and the reception window is closed, the controller sends a data burst of 9 packets containing its public key. The public key length is 32 bytes, each data packet can hold up to 4 bytes, so in the end there are 8 packets containing data plus an additional packet containing the checksum of the previous 8 packets.

Typically, in case of failure in the checksum due to a missing packet, the burst is retransmitted until receiving the Burst ACK. In this test all burst were received at first attempt, and therefore there are no retransmission. In the case there is a failure, sender will enter an ACK timeout state and retransmit the entire burst. There is no NACK functionality or a way to detect which packet got corrupted or did not reach destination.

Burst ACKs are a critical part in the communication. As there is no double ACK functionality to make the process faster, ACKs are always sent 4 times in a row to make sure they are received at least once. If by any case info is correctly received and all ACKs are missed, receiver, would go into the next state while transmitter would keep sending data burst. Both devices would no longer be able to contact each other, so a global timeout scheme was adopted in each communication stage. If a communication stage takes 3x times longer than usual, all the process will start over. They would have to start from scratch, but it gives them an additional opportunity to send to information correctly.

The full process reset mechanism is just an extreme solution to a very infrequent situation. In normal tests this circumstance only happens if provoked manually (by interrupting the line of sight), as usually all packets are correctly received the first time once the handshaking has been done.

After receiving the controller’s PK, the receiver sends his using the exact same routine. The controller receives it, ACK it, and immediately starts sending the Access Point credentials in a single burst. This transmission was optimized to make it fit in 8 data packets.

SSID is included in the first 4 packets, in clear, while the PSK is encrypted in the next 4 data packets. The Final Data packet will include the checksum of the previous 8 payloads.

The whole communication process of the real test that is reflected in the diagram took less than **4,5 seconds to complete**. As we can appreciate, the slowest and most inefficient part in the handshaking, as it suffers retransmissions and collisions and it becomes the slowest part while being the stage in which less information is actually transmitted. In fact, a whole Data burst of 9 packets and its ACKs take around 30-40% less time to be sent.

4th (LAST) Working Version Transmission Diagram

The current working version, which is the fourth one, developed in September 2019, reduces the delay in an additional 25% compared with the previous version. Its full transmission diagram is the following:

Figure 14 - 4th (Current) Working Version Transmission Detailed Diagram

This implementation only differs in the total number of data bursts to be sent. In the 3rd version, a dedicated key exchange procedure was implemented, so in the end, the controller would send two data bursts of 8+1 packets each. This design is completely logical for most of the cases. However, in this case, in which little real information is being encrypted and transmitted (only 32 bytes), it is inefficient to create two bursts and send them independently.

This 4th version suppresses the controller's public key exchange burst and includes the 32 bytes of the key inside the data burst. This saves a lot of time, as the most inefficient moment of the transmission is the ACK window, in which a same small piece of information is exchanged several times during a large window of time to guarantee confirmation will arrive.

Also, there the shared key calculation and information decrypting were moved to the moment after the end of the transmission in the IoT Node. This decision saved further time in the communication, as key calculation and information decrypting can take up to 0.5 seconds. In the controller, however, shared key calculation must be done just after receiving the node's public key, as the shared key will be immediately used to encrypt the information before sending it. Therefore, this implies losing 0.4-0.5 seconds that end up delaying the whole transmission process.

The improvements in the version 4th achieve a time of **3.7 seconds** until receiving the Data ACK in the controller. This is a reduction of 0.75 seconds compared to the previous version, which is a 15% less time.

An additional 0.4 seconds are needed later in the receiver to treat the information, but this step does not belong to the communication itself, as the controller is no longer involved in this stage.

If the information is sent in clear, there is no time savings as there are no keys involved and the communication process is almost identical to the 3rd version.

Time Evolution Analysis

The delay in the in the transmission is one of the most important variables to take into account. The transmission delay for a secure key injection system via infrared can turn a useful design to not be commercially possible as an end user would not accept spending too much time in such a simple process. There has been put a lot of work into reducing this delay as much as possible. Below we can see a comparison figure of different time graphs and the efficiency increase over each new version.

All the times reflected were extracted from the Receiver communication routine in different moments of different versions of the protocol development. They take as reference the time in which the first New Hello packet from the Controller arrives to the Receiver.

Just to have an initial reference, the 1st working version, which is not featured here due to its early stage and inefficiency, implemented a Stop&Go design, and it needed almost 10 seconds to transmit the information in clear. It was decided not to include a graph due to its extreme inefficiency.

We started analyzing the delays since the 2nd version. These three timelines belong to three different subversions from this 2nd working version. In it, the SSID and PSK were sent independently in different burst. Moreover, there was no encryption, and therefore there is no public key exchange procedure. This makes the delay shorter as information was sent in clear.

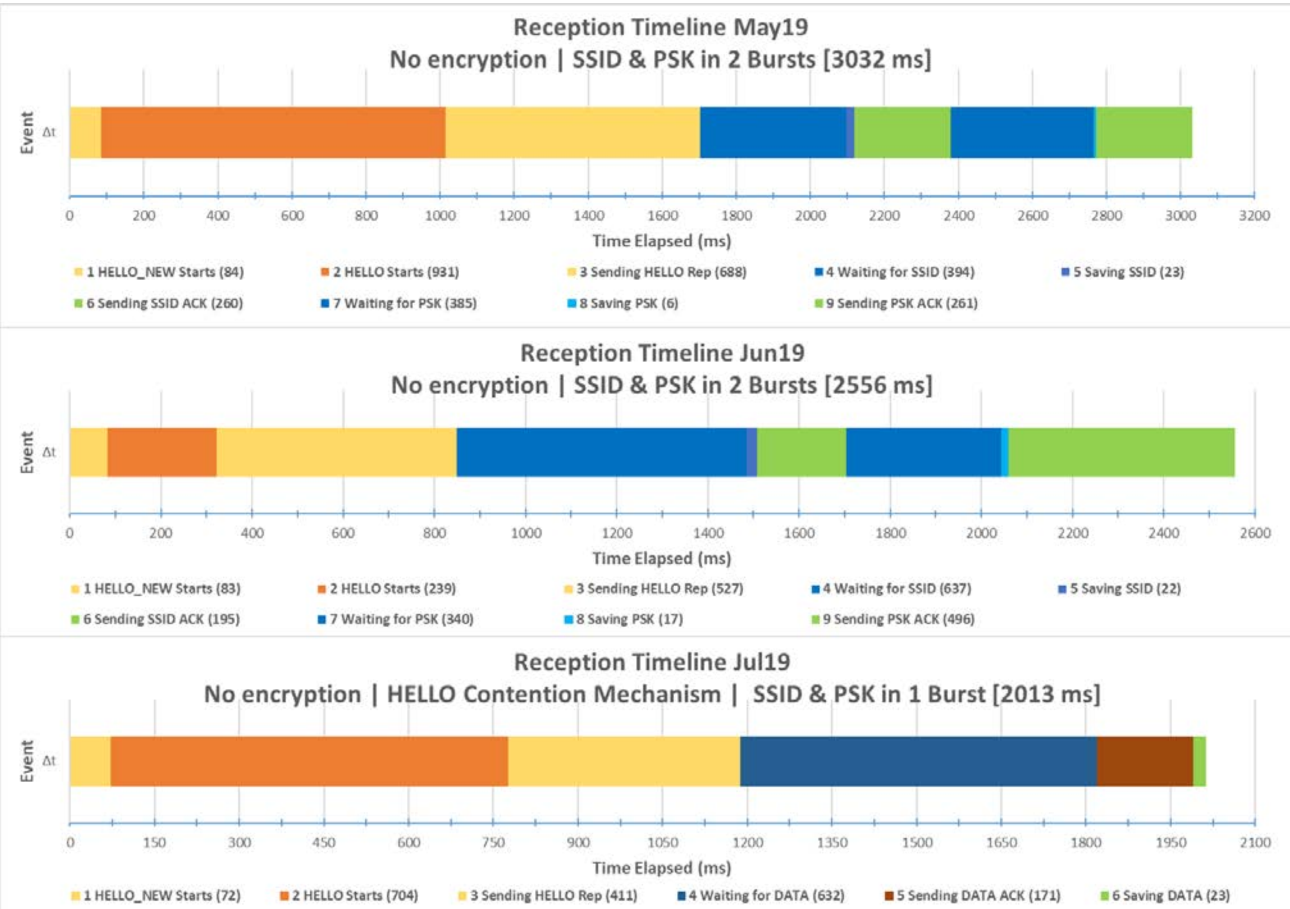

Figure 15 - 2nd Version Time Diagrams | Transmission in Clear

We can see in the previous three timelines the favourable evolution of the total transmission time. These three graphs are all from the same version with different adjustments.

We managed to go from 3 to 2 seconds, which supposes a 33% efficiency gain. This is due mainly to two different adjustments:

- IR RC6 Protocol modifications: The protocol parameters were customized to reduce blank periods and to increase its speed.
- Sending Logic: In the first 2 graphs, information of SSID and PSK is sent in two different blocks or bursts. This was changed in the third graph, were both SSID and PSK are included in a single larger burst. This greatly reduced times as we suppress one checksum and, more importantly, we now skip the time waiting for the additional burst final ACK.

However, none of the subversions from the 2nd version used encryption, as in that moment it was not implemented as speed and efficiency was the main concern. Once encryption was added, the times logically increased. The final results are the following:

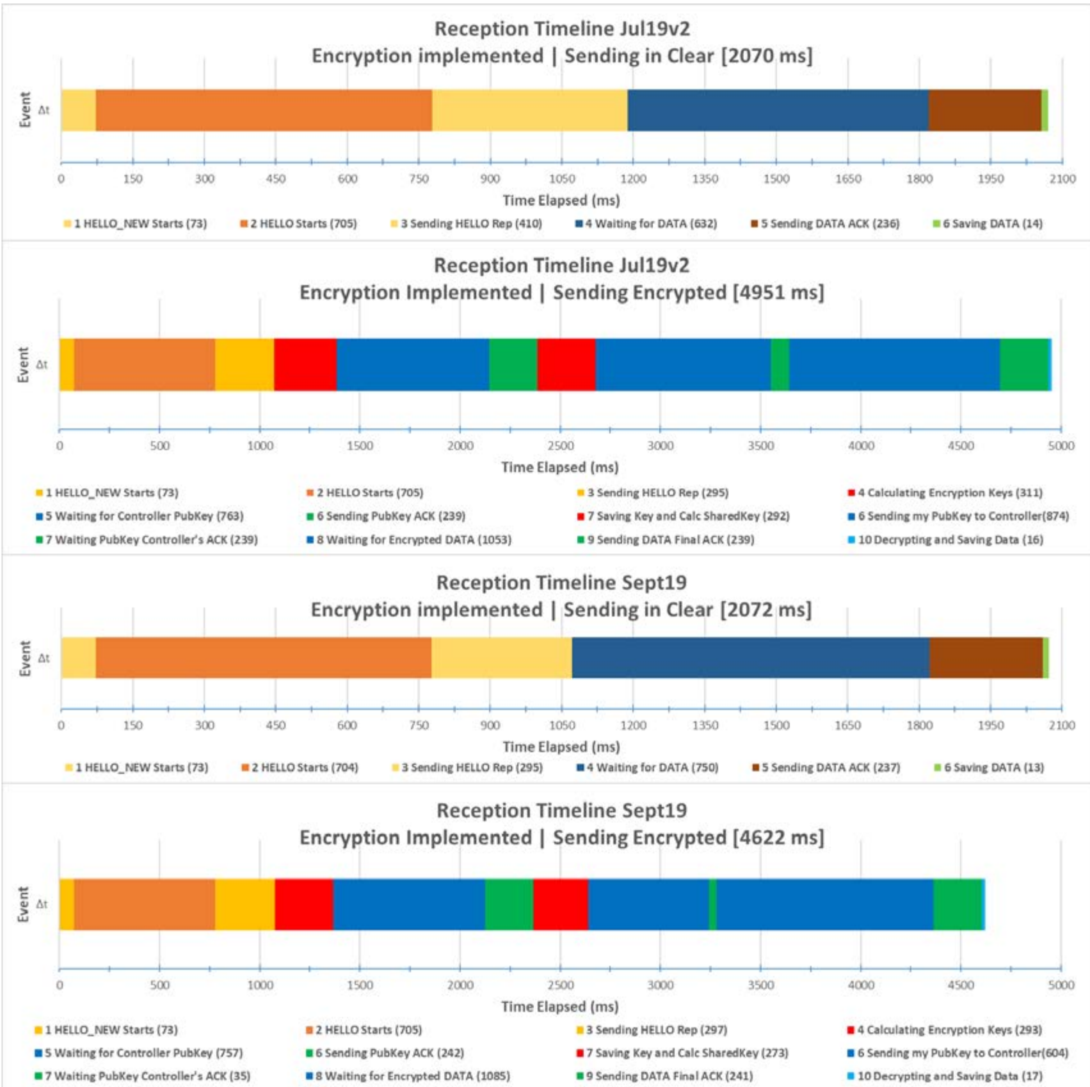

Figure 16 - 3rd Version Time Diagrams | Clear & Encrypted Transmission

Now, as encryption was added, we need to differentiate sending in clear versus sending encrypted. The times gathered from sending in clear are similar to the ones obtained in the third graph of the previous capture. We did not manage to make further improves in this part. However, a delay of two seconds can be considered a low interval, and, in practice, it is completely functional to be commercially implemented.

The 3rd working version, jul19v2, is the first one implementing an encrypted communication. It used to take up to 5 seconds to transmit, that is 1.5 times more delay than sending the information in clear, as we were sending 3 information bursts rather than one. A second iteration of this version, sept19, employs additional adjustments which optimize the way each block is sent and take advantage of empty time slots to make calculations instead of having the other device waiting. This reduced the time from 5 to almost 4.6 seconds, which is around an 8% efficiency gain.

Finally, the 4th and current version (sept19v2) was developed after analysing these time diagrams. It was concluded that the time cost of dedicating two independent bursts for the Controller to send both public encrypting key as well as the Access Point credentials was too high. Therefore, a single larger burst is now sent by the Controller consisting of 16+1 data packets.

Another modification was done in the IoT Receiver node, in which the shared key calculation and information decrypting is now done after concluding the transmission. This also saved some additional time.

In the case of transmitting the information in clear, there was no improvement as all the previous changes we related to the encryption key, which is no managed in this kind of transmission. Therefore, the time for transmitting in clear in the 4th version is the same as the time obtained in the 3rd one.

The commented results are summarized in the following two graphs:

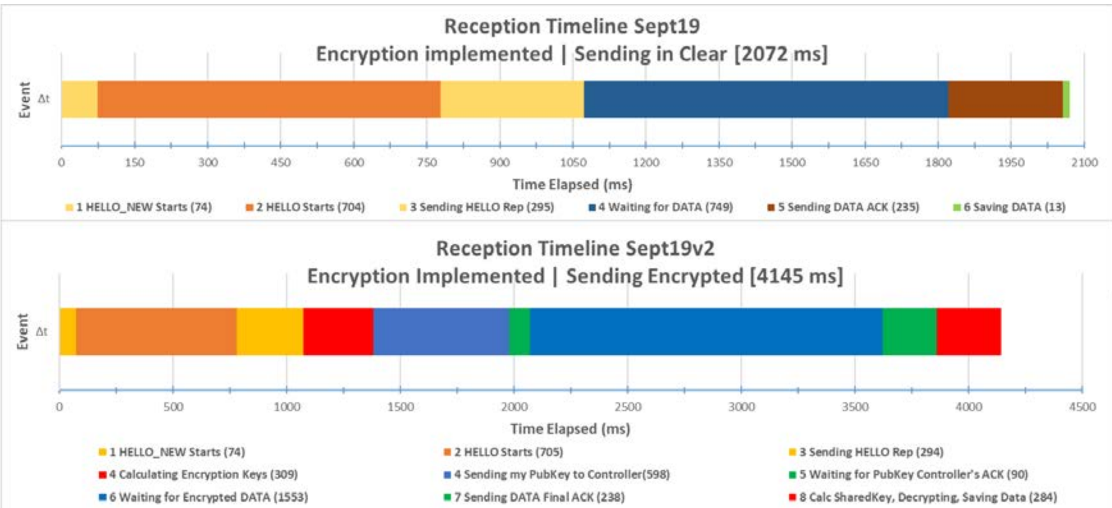

Figure 17 - 4th (Current) Version Time Diagrams | Clear & Encrypted Transmission

Although the total time appears to be 4145ms, the transmission actually ends in **3861ms**. The last red block is dedicated to encrypting key calculation a data decryption, that takes 284ms, but it should not be considered part of the transmission as the Controller is no longer involved at this point. Therefore, this version achieves a reduction of around 500-700 milliseconds in the transmission time, which translates to a 11-15% efficiency gain if compared with the previous version.

Finally, as a summary of all the time information explained in the section, an ultimate time graph is exposed. It reflects the evolution of the transmission time through the different programmed versions of the code.

The results are the following:

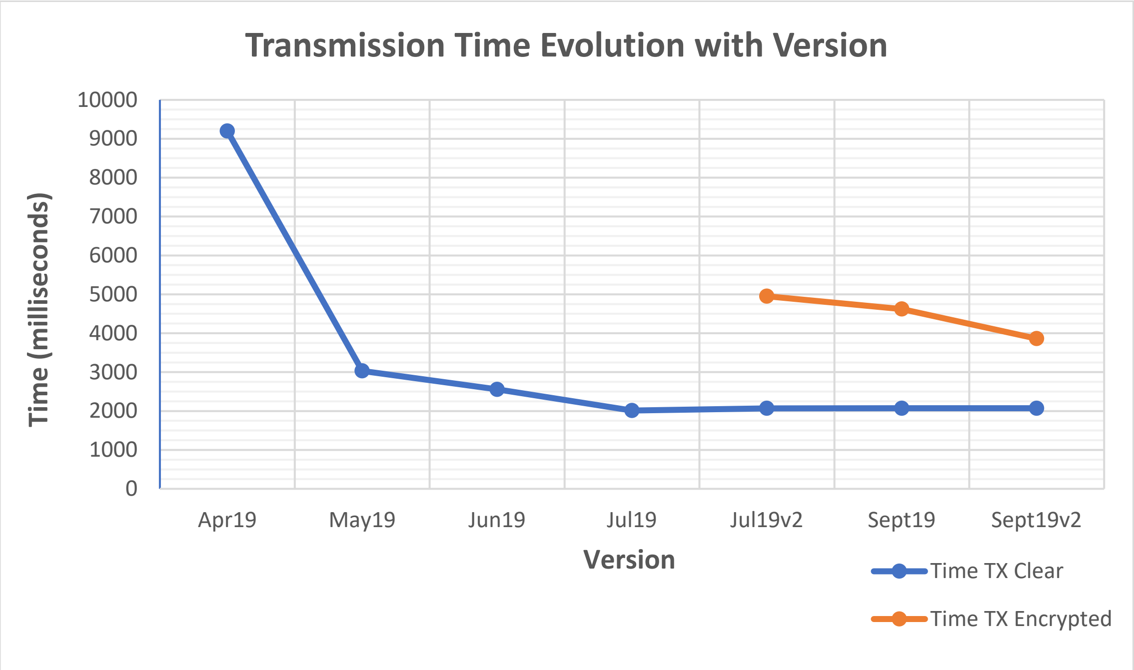

Figure 18 - Transmission Time Evolution with Version

As we can see, encryption was added once the transmission time in clear was an acceptable number. We can also see that information regarding the very first version (Apr19) is not included in any of the previous graphs due to its extreme inefficiency. However, it is interesting to include it in this graph as it was the initial reference point from which new versions where developed. This explains the jump from Apr19 to May19, where almost all the logic was changed.

The clear transmission also suffers a slight increase after adding encryption, as a consequence of the extra code. However, this increase was reasonable, about 50ms.

Regarding the encrypted transmission time evolution, first version was slightly above 5000ms. We managed to reduce it by more than 20% to almost 3850ms. Despite achieving the maximum efficiency level in the clear transmission, the encrypted variation might be improved further in the future with some advanced adjustments.

3.3.2. Raspberry Pi

The emulation of the IoT Access Point is done through the Raspberry Pi Model 3b+ device. The components and interfaces of the device were already described in the hardware part. This section will be focused on the configuration of the RaspPi that was done in order to get the device configured and working as an Access Point.

In contrast with the ESP8266, all the development has been done using the Python programming language. Due to library dependencies, some scripts use version 3.6, while the other one is using an older version, python 2.7.

3.3.2.1. Initial Configuration

The Raspberry comes without any kind of operative system. In fact, not even the SD card where the OS system should be installed is included. For this project, we are using the Debian based OS designed for this device: Raspbian.

Raspbian is similar to other Linux based OS, and all the configuration was done through terminal commands. It only differs from it in some configurations and the management of the security policies, which have to be disabled locally. For example, in order to get the RaspPi camera module running, or to allow remote SSH connectivity, we would need to modify these security policies first.

Once security policies are configured, it is important to set a static IP within the domestic network, in order to access it remotely later by other computers and to facilitate the configuration of the routing table once the Access Point is running.

This is done by modifying the file ***/etc/dhcpcd.conf***, and adding, for example, the following information:

```
interface eth0
        static ip_address=192.168.1.11/24
        static routers=192.168.1.1
        static domain_name_servers=192.168.1.1
interface wlan0
        static ip_address=192.168.100.1/24
        nohook wpa_supplicant
```

In this case, eth0 corresponds to the wired interface which connects the device directly to the home internet router. This is the one which will actually be used when configuring remotely the device from a computer.

On the other hand, the wlan0 interface will be dedicated later as an Access Point. Therefore, we just assign it another static IP, it ends in .1 as it will act as the gateway for the IoT devices connected to this wireless network. However, wlan0 configuration does not have to be added manually, as the AP configuration script does it automatically when executed.

In the project, the programming, execution and debugging of the RaspPi code has been completely done in remote. The computer used was running PyCharm Python IDE on a Win10 OS. This IDE has support for remote Python Interpreters and automatic file remote loading, easing significantly the programming of the device.

3.3.2.2. Access Point Automatization Script

The creation of the Access Point can be easily done using the terminal. However, it takes a significant number of steps to do and it is crucial for the project to provide an easy way of configuring it automatically and overwriting a previous running AP.

The configuration was done using Python 3. It uses already existing local libraries of python: *os*, *sys*, *subprocess* and *re*, so no external packages have to be installed beforehand. However, in order to make the routing IP tables persistent, an external package was installed: *iptables-persistent*. Other packages, *dnsmask* and *hostapd*, have to be installed as well. However, the installation of all needed packets has been automated in the script.

The script has been called wifi_setup.py. It takes two parameters as arguments: SSID and the PSK that the configured AP will have. In order to get it running, it should be called via terminal with:

```
sudo python3 wifi_setup.py <SSID> <Key>
```

If sudo is not provided, the script will try to recall itself using sudo if possible.

The script does the following actions:

- Installation of all needed packages for the script to work properly.
- Destruction of previous configuration of an Access Point if it exists.
- Configuration of the new AP, with the provided SSID and Password.
- Routing configuration between eth0 and wlan0 interfaces. It allows connectivity to internet of the devices connected to wlan0 by default, through NAT Masquerading.
- Persistence configuration: The configuration was designed in order to assure persistence in case of device rebooting. The AP should run automatically every time the device is powered on.

As the script is included in the **Annex III**, and by a simple read it can be easily verified which files are modified in order to make the configuration, we are not including an extended explanation here. The relevant files involved in the AP configuration are the following:

- ***/etc/dhcpcd.conf***: It assigns static IP to the RaspPi for the wlan0 interface.
- ***/etc/dnsmask.conf***: It defines the range of address for DHCP assignation. However, all IoT devices will connect with a static address. Addresses from .200 to .250 are reserved of dynamic assignation, in case the user wants to connect another device without a static IP.
- ***/etc/hostapd.conf***: This is where the information related to the AP is kept, such as SSID, PSK, channel, mode, interface, etc.
- ***/etc/iptables/rules.v4:*** This file hosts the information regarding the iptables configuration to be restored when it boots up. This file is created automatically by iptables-persistent package, and it is overwritten each time the script is called.
- ***/etc/rc.local***: This file includes a set of commands to be executed every time the device boots up. It is modified to load the iptables rules.

3.3.2.3. QR Scanner Script

The second programmed script is a tool which facilitates the act of transmitting the AP credentials from the IoT Controller to the RaspPi. Although this can be done using the previous script and manually inserting the SSID and WPA2 Key, this process is not agile or satisfactory for the user.

Therefore, we propose using this program as it takes out much of the manual trouble, as it only requires showing the camera a valid QR code with a SSID and PSK. Moreover, this script is logically linked with the AP creation one (which is the one we just explained).

The program uses OpenCV 3.4.4 to recognize and manipulate pixels of images, in order to look for a QR Code. As Raspbian does not include this library by default, it has to be compiled and installed before being able to run the scanner.

The installation of this library is quite a long and complex process[20b], and it can even take several hours to complete. Nonetheless, there are several tutorials available online, such as the one I used in this case.

Apart from OpenCV, there is another external library which must be installed: zbar. This library will be the one in charge of analyzing the image captured by OpenCV to look for a valid QR Code. It can be installed in the system, with `apt install python-zbar`; or using pip, with `pip3 install pyzbar`.

This script uses Python 2.7 instead of Python 3. The main reason for this choice is the lack of documentation for OpenCV, which is quite an extensive library, in the later version. The developed code is a modification of a script found on the official Raspberry forum[21b].

The implementation in the RaspPi forces the Scanner to keep taking and analysing pictures periodically, until a valid QR is found. A QR is valid if it can be correctly read, and if the decoded information matches the format used to encode the SSID and PSK. When these conditions are met, the program stops scanning and calls the wifi_setup.py script with the SSID and PSK as arguments, allowing it to configure the Access Point.

The camera scanner can be manually run with `python2 camera_scan.py`. However, the Raspberry is configured to run this script automatically in every bootup, unless there is an existing Access Point. In this case, the script has to be manually run.

Although the script, along with explanatory comments, is attached in the **Annex IV**, it is interesting to briefly discuss its execution flow, which can be found in the next figure:

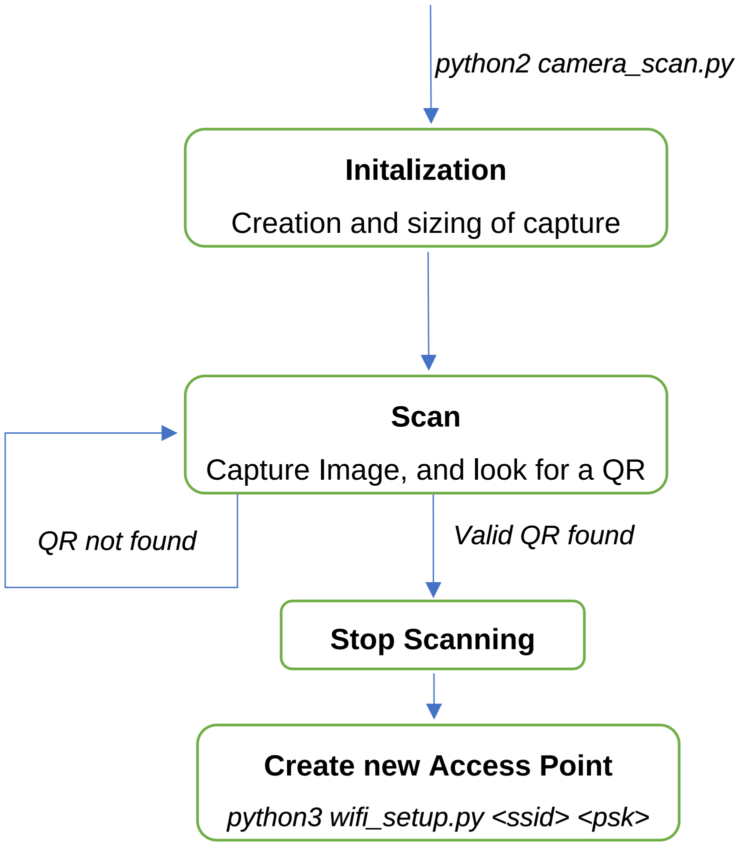

Figure 19 - QR Scanner Script State Diagram

When called, manually or automatically after booting up, the script initializes the RaspPi camera by specifying some capture configuration parameters, such as capture resolution. Once these parameters are set, the script enters in the next state: scanning.

The scanning phase is an endless loop in which it periodically retrieves captures and analyse them until finding a QR code. While the capture does not feature a valid QR Code, the script will ignore it and make a new one. Finally, when it gets a valid QR code, it stops scanning and calls the Access Point creator script *wifi_setup.py* with the parameters decode from the QR Code that was present in the last capture.

Currently, the program processes around 2-3 captures per second, with a resolution of 960x720 pixels. Increasing the frequency of the resolution could cause the device to heat up and to overload, so it is misadvised as it is intended to be permanently running in background.

Unfortunately, hardware limitations in both RaspPi camera module and Heltec Wi-Fi Kit 8 screen makes it currently not possible to directly scan the OLED screen with the camera module. However, it works if we use the higher resolution camera of our mobile to scan it, and then scan the code with the Raspberry from the smartphone's screen. There are many apps able to retrieve not only the decoded information from a QR Code, but also to represent that same QR code in the mobile's screen.

3.3.2.4. Serial Port Reader Script

As an additional alternative, a script able to automatically read the credentials from the serial port has also been implemented in the Raspberry Pi.

It was designed to work straightforward: first the script is run from a terminal, and secondly the IoT Controller is plugged in to the RaspPi via USB. The script will start reading directly and look in the Serial output for the Wi-Fi credentials: SSID and PSK. Once the credentials are read, the script will finish and call the wifi_setup.py program in order to get a new Access Point running, with the terminal command:

```
python3 serial_reader.py
```

This script works both with python 2 and python 3, however it has been thoughtfully tested only in python 3. It is also important to notice that, in order to get the correct behaviour, the IoT Controller must have assigned the port ttyUSB0. If there were any existing device connected to the RaspPi, it would have to be removed before connecting the controller.

The complexity of this script is very low, as there is a preinstalled library for python, called serial which handles all the communication from and to a serial port. However, for this application there is no need to send any information to the Controller device. Its full code is available for reading in the **Annex V**.

3.3.2.5. Access Point Initialization Summary

As multiple alternatives were proposed and implemented for configuring an Access Point in the RaspPi, it is important to put them all together to analyse how they interact with each other.

The following figure shows the three working possibilities for the user to configure the AP:

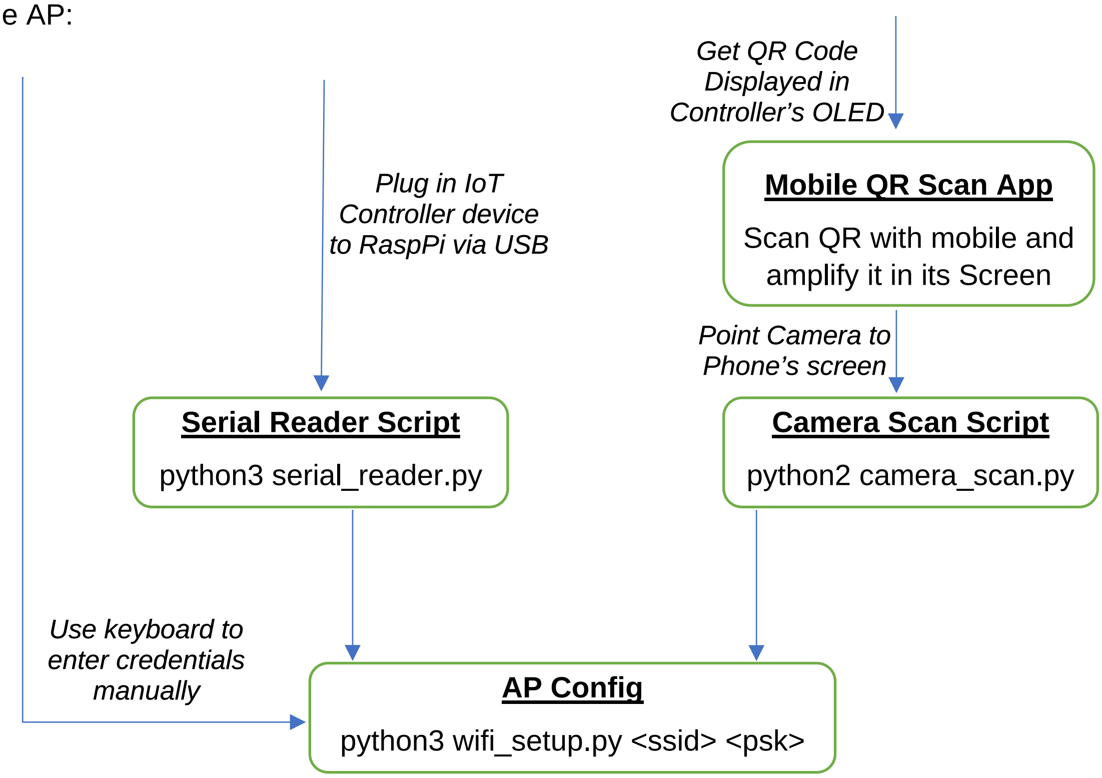

Figure 20 – Access Point Setup Alternatives in RaspPi

As featured in the diagram above, there are currently three different alternatives for having the access point configured. While the first one would simply be typing it manually, the other two makes the process easier for the user. These information extraction scripts could also be configured to run in background during the RaspPi bootup, as in a normal scenario a user may not have knowledge to access it using a remote session.

As we mentioned before, there is a fourth alternative not represented in the diagram, which consisted in transmitting the key to the RaspPi using infrared, as it is not working currently, it was simply omitted.

3.4. Developmental Setbacks

The development of the introduced solution to the proposed project has not gone straightforward, as many obstacles and unexpected situations have often come across. The following list is a summary of some of the most remarkable of those difficult scenarios and circumstances we had to face and solve in order to allow us to keep making progress in the project until the stage at which we have decided to stop its development, as a project is always subject to improvement and should never be considered to be finished.

ESP8266 Hardware Design

Initially, the IR Led and receiver seemed to work fine if directly connected to the board pins. The issue was that initial transmission tests were too simple to detect the failures. We took almost two months, when a more complex transmission protocol was developed, to realize that the LED current was not as high as desired and the high Bit Error Rate was causing too many packets to get corrupted.

Moreover, we spent some time until we realized that there are different NodeMCU boards that may look similar but have some differences, and in the end, the current supplied to their pins was slightly different, causing troubles when switching on the IR Led to transmit.

IR Transmission Protocol

This has always been the core issue of the development, and it would be more precise to consider it a large number of small interconnected issues than a single large one. Also, the previous point made the transmission protocol to be even more erratic until we solved the hardware issues.

The list of protocol issues is extensive, and many of them were already covered throughout the document: transforming a one way with no feedback protocol into a bidirectional acknowledged one, increasing the bitrate by modifying the RC6 protocol itself, dealing with the high BER by adding two transmission LEDs instead of just one, doing dozens of versions and iterations of the protocol, testing a huge number of approaches until deciding which version worked best, etc.

Communication from Device to the User

This was another important issue. As the SSID and PSK are generated internally by the controller, the user has no direct access to this information. Accessing it through a cable is not an option as the controller was conceived to be mobile, and it should be able to be accessible from everywhere.

With these circumstances in mind, we finally decided to use a different board as the Controller, one which had an OLED built-in screen, such as the Wi-Fi Kit 8. However, as it can be considered to work, screen pixels turned out to be quite tiny, and in the end the scanning of the QR Code cannot be considered to be a satisfying experience, as one ends up struggling with the camera focus and distance until the mobile finally decodes the QR. Therefore, this issue can be considered to be partially solved, as it would be interesting to try a different screen with larger pixels and check if the scanning is easier.

Design of the IoT Router

We would have preferred to use a commercial model as the IoT Router, as it would have supposed a more realistic scenario for the project. However, the lack of programmable commercial routers and their high price made us finally choose a Raspberry Pi 3b+, due to its versatility, processing power, and ability to customize.

IR Communication from Controller to IoT Router

IR communication between Controller and IoT Router was our initial idea to transmit the AP credentials to the Router. However, these two devices are completely different, and after reading a lot of documentation and intensive testing, we decided to look for alternatives. The issue here is developing a protocol for the router that should be completely compatible with the one the Controller already uses. Moreover, time it has to be fully transparent for it as well, which means that the controller would connect and transmit the information to the router exactly the same way it does it with any IoT Node. We finally implemented the QR scanner script as the alternative for loading the credentials without having to manually type them in the router.

4. Results

Although all the project results have already been exposed and analysed in the previous section, they are scattered throughout that development section, making it hard to retrieve them all at a glance.

Therefore, this part will briefly collect all those dispersed results, conclusions and comments about them, serving as a global summary of what the project outcome was.

4.1. Hardware Adopted

The designs and equipment that were finally chosen, among the different alternatives already exposed, are the following:

- Acting as any IoT Node, we finally chose an ESP8266 chip based NodeMCU development board, AI Thinker version. It has connected an IR Receiver model TL1838B directly to a GPIO Pin and two IR LED emitters in series using an amplifier circuit, as shown in figure 6. After intensive testing, we found out that LEDs were not receiving enough current, making the Bit Error Rate too high and losing many packets.
- To act as the IoT Controller, the ESP8266 chip based Heltec Wi-Fi Kit 8 was finally selected. Both OLED screen and battery port are used in practice, and we only soldered an IR LED emitter and an IR Receiver model TL1838B directly connected to the board GPIO PINs. There was no need for an amplifier circuit design for the LED, as the current supplied to the PINs is higher than in a NodeMCU board.
- The IoT Router and Firewall functionalities are finally implemented by a Raspberry Pi Model 3b+. Among all the alternatives used for loading the Private Wi-Fi Key, we are finally adopting the one that uses the USB Serial Port for a single reason: it does not require any additional hardware or circuit, which is translated into a reduction of both cost and complexity.

In a real scenario, we would only need one IoT Controller and a single IoT Router. The number of IoT Nodes, and its associated components, will depend on how many IoT devices we would like to emulate. To make a Proof of Concept, a single IoT Node is enough to get it working.

The analysis of cost for all the components that were selected to have all the equipment working will be discussed in the budget section.

4.2. Software Adopted

During the accomplishment of this project, the software has been in constant evolution and improvement. Many different versions were developed and tested for all the different devices of the scenario. The followings are the ones that were selected in last instance and, therefore, are running in the devices:

- Each IoT Node will implement the latest version of the receiver C code, coded in September 2019. The full code is available in the **Annex II**, as well as its documentation, which is attached to this report.
- Analogously, the IoT Controller implements a similar code than the IoT Node, based on the most recent version of the controller C code.
- The IoT Router firstly needed to get configured, following the steps detailed in the section 3.3.2.1. Once configured, due to the hardware finally selected, it will only implement two of the scripts programmed in Python 3:
    - Access Point Automatization Script: wifi_setup.py, which creates and configures a persistent access point based on the SSID and PSK passed, respectively, as first and second parameters.
    - Serial Port Reader Script: serial_reader.py, which waits for the IoT Controller to be connected to the USB Port and reads the serial outcome until reading the SSID and PSK, moment in which it will call the wifi_setup.py script to establish an access point automatically.

All the developed code, including other Python scripts that are featured in the Development section but not here, are included in the Annex for reader's consulting.

4.3. Private Key Transmission Results

The delay suffered as a consequence of transmitting the Wi-Fi private key, from the IoT Controller, which previously randomly generated it, along its associated SSID, towards an IoT Node, and using the last versions of the transmission protocol and the hardware described above, is the following:

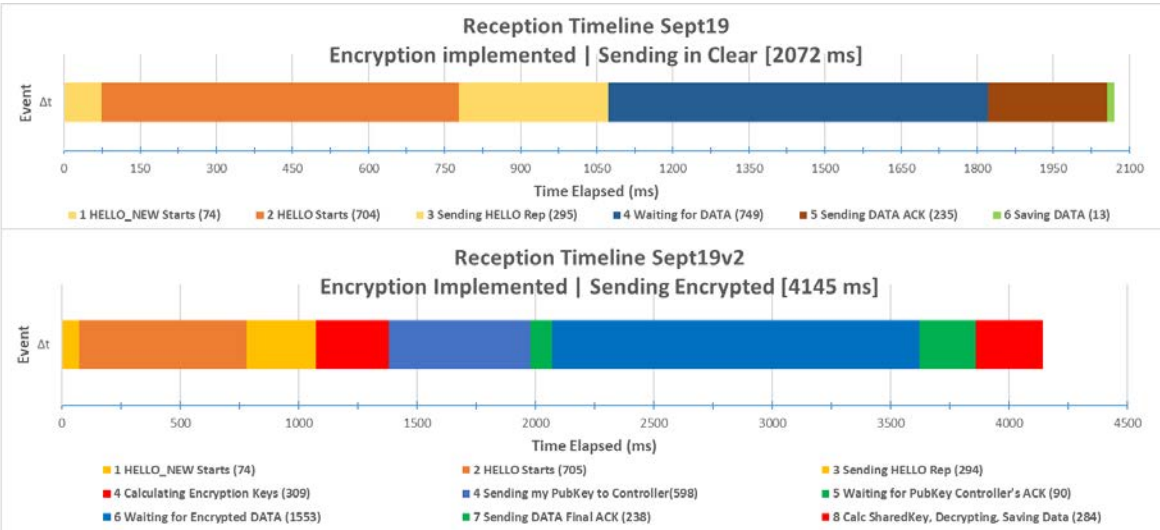

Figure 16 - 4th (Current) Version Time Diagrams | Clear & Encrypted Transmission

It roughly took 2 seconds to transmit both pieces of information in clear.

In case of the encrypted transmission, it took 3,86 seconds to transmit them, as there were around 0,3 seconds used later by the receiver to process the information, but that part does not belong to the transmission itself, which was over at t=3,86s.

Both delays make the use of infrared as the technology to transmit the key as a commercially viable safe alternative for domestic and closed environment use. Although a delay of 3,86 is significant, in a practice scenario it is a value that can be tolerated, as this step is only performed once per device, and usually there are not many domestic IoT devices that have to be configured altogether.

Moreover, in case a massive configuration has to be quickly done, information can also be transmitted in clear, reducing the transmission time in a 50%. Infrared should keep the information secured as long as there are not third persons in the line of sight.

4.4. Key Loading into the IoT Router

Several alternatives, consisting in some cases in just software, and in some others in both hardware and software, were proposed:

- Manual Key Loading: By directly using the wifi_setup.py script locally or remotely by manually inserting the credentials in a terminal. It works, although it is not user friendly.
- QR Code Scanner: By using the camera_scan.py. It works for normal size QR codes. However, due to the tiny size of the IoT Controller OLED screen, it is not able to read from it directly, which makes it an unviable commercial solution until a solution is found. Also, it needs a RaspPi Camera module, which is as expensive as the Raspberry Pi Model 3b+ itself.
- Infrared Credential Injection: It would basically imitate the concept used in the communication between the IoT Controller and the IoT Node, but in this case the controller would communicate using infrareds directly with the IoT Router. However, there are some technical difficulties that we were not able to save, as the libraries and hardware are completely different for both devices. This option, although technically possible, it would be extremely hard to implement and not worth doing it. Moreover, it requires some extra hardware which would raise the costs and complexity.
- Serial Port Scanner: By using the serial_reader.py script. This script could just be configured to run on system boot and keep listening for a device to be attached to the USB Port. It is the simplest and easiest solution, although it requires physical access to the Router which should not be a problem in most of the cases. This option is the preferred one as it is cost-free and commercially viable.

Among all the options, the last of them, the serial port scanner, is the one we are finally using due to the reasons exposed. It is the simplest, the most robust one and also the fastest. It is able to read the credentials from the IoT Controller in almost no time. In terms of security, it is safe as information is sent using a physical wire. Moreover, it also reduces the overall cost of the solution by not including additional hardware; and finally, it is user friendly, as it only requires a human for plugging in the device via USB to the Router.

Therefore, this part of the project was also successfully solved, although some of the solutions could not finally be developed to get them working. Once the AP is running and the IoT Nodes are connected to it via Wi-Fi, the project becomes out of further scope as the interaction between both devices and between the IoT Node and external servers in the Internet would depend on the device application itself and on the user intention towards the device.

5. Future Development and Conclusions

5.1. Future Development

At this point, every functionality which was developed or implemented has already been covered in the previous sections. However, this project should not be considered as totally completed, as there are many details and approaches with room for improvement. Despite implementing many of the core features, there have been left behind some other secondary features that, in the end, would suppose a huge difference in terms of overall usability, performance and security.

This chapter tries to briefly cover all those features and ideas that were omitted due to time restrictions, technical difficulty or any other circumstance. It also serves as a critical review of some of the already implemented features which might be reiterated in order to raise the quality and make the whole project suitable as a commercial product.

IoT Device Application Layer and Custom Hardware

Despite the core idea of the project was about IR secure transmission protocols, the hardware acting as an IoT device is just a mere actor serving as an example of how an IoT node is like. Therefore, the device used for this project has no functionality included by default, which basically means that it would act as a dummy node.

It would be a great improvement to build a small application layer that give support to some of the most common applications these devices usually implement, such as weather analysis, light analysis, status monitoring of some other device, etc. However, to make a correct software implementation of an IoT Application Layer which uses this project's basis as the mechanism to safely retrieve the local network credentials it would also be crucial to design the hardware, a custom board, which could be based in the ESP8266 or in any other chip, as well as all the board schematic and PINs. The board should be easy to setup for the most common IoT applications, and at the same time it should allow the user to fully customize it for any other more complex use.

Due to the complexity of those upgrades, this section could be perfectly separated into a completely different Internet of Things project. In the end, it involves both hardware and software aspects, and its size would make it more appropriate for a doctorate thesis than a master's.

IoT Custom Router and Application Layer

Similarly to the previous point, our IoT router is not really a commercial router, but a Raspberry Pi configured to serve as one.

The idea is basically the same, develop from scratch a programmable router whose firmware already include some sort of IoT application layer, with support for custom firewall configuration for non-technical users as well as a useful IoT device manager.

The complexity of this implementation should not be underestimated, as the hardware would be needed to be designed from scratch, and the software should include a custom firmware with routing, firewalling, and support for being programmed. In the end, it would be similar to a diminished version of the RaspPi we using, removing all the features de RaspPi has and that are not needed for this project, and reducing its cost be using cheaper less powerful hardware, and implementing some kind of IoT management application.

Again, this implementation would be too large to be carried on by a single person and it should be considered as a completely independent project that would take a lot of time to fully implement.

IR Transmission Protocol Improvements

One of the parts of the project in which more time was invested is the transmission protocol. Many different versions were developed, and extensive testing was done until the last version provided in this document was delivered. However, this last version should not be considered as the final or the definitive one.

Some parts of the protocol have room for improvement. With the correct ideas, some additional time in the transmission could be saved, although the current delay is close to the theoretical limit. Nonetheless, time is not the only concern. There are some other features equally important, as the user experience, which was not heavily developed. Also, it would be interesting to work on versions for other platforms, or even to develop our own IR library to avoid depending from third parties at the same time we increase its efficiency.

IR Communication with the Router

It would be useful to allow the IoT controller to directly transmit the credentials using the existing protocol, or a modification of it, to the router. In the project, however, it was not possible to implement it as the router is a completely different platform and there was no way to make the RaspPi IR library compatible with the ESP8266 IRRemote library.

In order to make this possible, it would be needed to develop an IR library in the RaspPi with the same protocol specifications and timings than the one in the IoT Controller. This is not a trivial implementation and it would require advanced knowledge as well as a lot of time for implementing and testing.

QR Code Scanner Enhancement

The QR Code currently serves as the one way of transmitting the randomly generated credentials to the user. However, the OLED screen used for its display has low resolution and too tiny pixels, making it hard for any camera to decode the information. In fact, currently the RaspPi camera module is not able to scan it, and the only way for it to scan is using the phone camera (for scanning from ESP8266) and screen (for representing the QR bigger) as intermediary.

In this case, the software implementation is correct, and the issue lies in the size of the OLED screen that the Wi-Fi Kit 8 board has. An alternative would be using another board and welding a bigger screen module to it. If we manage to make the pixels a bit bigger, scanner would theoretically be able to detect the QR and decode the information. However, this needs to be done and tested in practice that confirm this hypothesis.

These are some of the possibilities in the future development of the project, from its current stage. However, due to the project potential, there is an almost endless list of possibilities and upgrades and, in the end, this could be refined to make it commercially viable and profitable, as it would be a very useful ecosystem for many people.

5.2. Conclusions

Despite not being covered by any existing research paper, the act of using infrared technology to increase the security in the setup process of an IoT device in a domestic ecosystem has been proven to be not only possible, but realizable in commercial terms. Throughout this research there has been many struggles to make the solution efficient and user friendly.

As an additional layer, a key exchange protocol has also been successfully programmed into the IR transmission protocols. Moreover, the hardware and circuit designs are not complex or costly, as we will see later on in the budget chapter.

Although achieving positive results, all the project was based in a hypothetical scenario in which all the devices are representations of how a real IoT device would behave, and, in the end, they do not have any working application layer implemented. The next necessary step would be to design from scratch a real IoT device which already incorporates an IR transmitter and receiver module, along with the transmission protocols that has been programmed.

This project also puts the focus in the lack of privacy current IoT users suffer at their homes. Private information is often gathered and sent by these devices to the manufacturer servers, without user knowledge. In other cases, external servers are programmed to scan for these specific devices to extract user habits and preferences.

In order to address the privacy issue, our proposal suggest to setup a distributed network, based on the operating principles of the existing TOR network, in which every packet sent to the network is iterated randomly by other nodes in the network to anonymize it. For this purpose, the suggestion is isolating all the IoT devices under an IoT router acting as the gateway and firewall for all these packets.

In the project, the role of the IoT Router is falls in a Raspberry Pi, whose Raspbian operating system allows deep customization and routing capabilities. To enhance the RaspPi behavior, some custom scripts were programmed to ease the user interaction with it, helping him to configure new Access Points in a pseudo-autonomous and automated way.

The RaspPi worked perfectly for this task, however, it would be interesting to invest on designing a cheaper device that could perform properly without being that expensive.

Domestic IoT is an investigation field which has not been covered in depth. In the last years many advances have been made in making IoT useful and user friendly. However, other more important but less tangible principles have been left behind, such as security and privacy. This project offered an untapped alternative to address these two concerns. With further research and investment, it would be possible to develop a commercial patented version which serves as the confirmation of the viability of this research.

6. Budget

This project contemplated both hardware and software design necessary to provide a solution to the introduced problem: safely transmitting encrypted Wi-Fi credentials using two IoT devices and infrared technology.

The costs involved in all the hardware are summarized in the following table:

Hardware Cost Analysis		
Item	Quantity	Unitary Cost (VAT excl.)
Heltec Wifi Kit 8 ESP8266	1	5,048 €
NodeMCU V3 ESP8266 AI Thinker	2	1,588 €
Raspberry Pi Model 3b+	1	29,578 €
*Raspberry Camera Module	1	21,298 €
IR Led Transmitter	10	0,144 €
IR Receiver TL1838B	6	0,034 €
BC337 NPN Transistor	5	0,006 €
*2N2222 NPN Transistor	3	0,007 €
Resistor 1kΩ±5%	5	0,001 €
Resistor 1.8Ω±5%	5	0,001 €
Resistor 5.6Ω±5%	5	0,001 €
*Resistor 22Ω±5%	5	0,001 €
*Resistor 10kΩ±5%	3	0,001 €
	TOTAL (with optional items)	**60,809 €**
	TOTAL (without optional items)	**39,537 €**
*Item is optional, as final design does not require it		

Table 9 - Hardware Cost Analysis

Smaller components quantity has been made redundant in case some of them break, as their cost is neglectable compared to the bigger ones.

However, the project was mostly based in research and software. As we can see in the following table, most part of the time has been invested in developing the code to provide all the different functionalities for every device:

Dedicated Hours Analysis		
Task	# Hours	Cost per Hour
IoT Controller & Receivers Hardware design	8	12,00 €
IoT Controller & Receivers Assembling	4	12,00 €
Raspberry Pi Hardware design	2	12,00 €
Raspberry Pi Assembling	1	12,00 €
IoT Controller Programming & Testing	150	12,00 €
IoT Receiver Programming & Testing	100	12,00 €
Raspberry Pi Setup, Programming & Testing	50	12,00 €
	TOTAL	**3.780,00 €**

Table 10 - Dedicated Hours Analysis

This table only includes the time put into developing the functionalities described in the project. It does not include the time invested in researching the current IoT situation, or the time that took writing this report. The hours should not be taken as an exact quantity, but as an approximation of the working time in each part.

When comparing both hardware design and software programming times, it is clear that the real challenge of the project was creating all the logic for every different system. Nonetheless, the hardware part is also as important, although we decided to employ self-contained development boards rather than soldering PINs and small chips to save time for the coding part.

Bibliography

Papers

[1] Diffie, Whitfield; Hellman, Martin E. 1976. p644 - p654. New Directions in Cryptography. IEEE Transactions on Information Theory.

[2] Mathy Vanhoef and Frank Piessens. 2017. Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2. In Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS). ACM.

[3] Indra Dwi Rianto. 2013. Anticipating WPS PIN Vulnerability to Secure Wireless Network. Computer Science Department, School of Computer Science, Binus University.

[4] Aparicio Carranza, Josue Magallanes, Casimer DeCusatis and Javier Espinal. 2017. Automated Wireless Network Penetration Testing Using Wifite and Reaver. The New York City College of Technology, New York.

[5] Angela M. Lonzetta, Peter Cope, Joseph Campbell, Bassam J. Mohd, Thaier Hayajneh. 2018. Security Vulnerabilities in Bluetooth Technology as Used in IoT. Fordham Center for Cybersecurity, Fordham University, New York, NY; Computer Science Department, New York Institute of Technology, Old Westbury, NY; Computer Engineering Department, Hashemite University, Zarqa, Jordan.

[6] Huichen Lin and Neil W. Bergmann. 2016. IoT Privacy and Security Challenges for Smart Home Environments. School of Information Technology and Electrical Engineering, University of Queensland, Brisbane, Australia.

[7] Vijay Sivaraman, Hassan Habibi Gharakheili, Arun Vishwanath, Roksana Boreli, Olivier Mehani. 2015. Network-Level Security and Privacy Control for Smart-Home IoT Devices. University of New South Wales. IBM Research Australia. NICTA Australia.

[8] McCoy D., Bauer K., Grunwald D., Kohno T., Sicker D. 2008. Shining Light in Dark Places: Understanding the Tor Network. In: Borisov N., Goldberg I. (eds) Privacy Enhancing Technologies. PETS 2008. Lecture Notes in Computer Science, vol 5134. Springer, Berlin, Heidelberg.

[9] Simon K., Moucha C. and Keller J. 2016. Contactless Vulnerability Analysis Using Google and Shodan. In: Foresti S., Persiano G. (eds) Cryptology and Network Security. CANS 2016. Lecture Notes in Computer Science, vol 10052. Springer, Cham.

[10] Ramzi A. Haraty and Bassam Zantout. 2014. The TOR Data Communication System. Journal of Communications and Networks, vol. 16, no. 4.

[11] Béla Genge and Calin Enachescu. 2015. ShoVAT: Shodan-based vulnerability assessment tool for Internet-facing services. Petru Maior University of Tirgu Mures, Dept. of Informatics, Tirgu Mures, Romania.

Online Sources

[1b] https://developer.ibm.com/articles/iot-anatomy-iot-malware-attack/ - 'Provision the Device' section

[2b] https://www.industrysafe.com/blog/risk-matrix-calculations-severity-probability-and-risk-assessment/

[3b] https://www.electronicdesign.com/industrial-automation/security-manufacturing-closing-backdoor-iot-products

[4b] https://hsm.utimaco.com/solutions/applications/key-injection/

[5b] USA Department of Homeland Security – WPS Brute Force Attack Vulnerability

[6b] Bluetooth Diffie-Hellman Vulnerability Note VU#304725

[7b] Bluetooth BlueBorne set of vulnerabilities Note VU#240311

[8b] https://www.connexionfrance.com/French-news/Smart-Monsieur-Cuisine-Connect-mixer-from-Lidl-France-has-secret-microphone

[9b] https://www.torproject.org/

[10b] https://www.shodan.io/

[11b] https://www.ahirlabs.com/2017/10/21/what-is-nodemcu-esp8266/

[12b] https://www.openimpulse.com/blog/products-page/product-category/tl1838-infrared-receiver/

[13b] https://uk.farnell.com/vishay/tsop2238/photodiode-ir-receiver-35m-950nm/dp/4913073

[14b] https://heltec.org/project/wifi-kit-8/

[15b] https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/

[16b] https://github.com/adafruit/Adafruit_SSD1306

[17b] https://github.com/nayuki/QR-Code-generator

[18b] https://github.com/esp8266/Arduino/blob/master/libraries/ESP8266WiFi/src/WiFiClient.h

[19b] https://github.com/agl/curve25519-donna

[20b] https://www.learnopencv.com/install-opencv-3-4-4-on-raspberry-pi/

[21b] https://www.raspberrypi.org/forums/viewtopic.php?t=76674

Annex

Annex I - ESP8266 Controller Code

```
/**
  * \defgroup main main.cpp TFM Controller Code
  * 2019 - Rafael de la Rosa Ortiz \n1
  * Allows a ESP8266 to transmit a <SSID,PK> to an IoT node.\n
  * It generates a random SSID and Key, using true random numbers, and saves it in its
EEPROM.\n
  * It allows to send the Key to a node securely by implementing a Diffie Hellman Key
Exchange Protocol.\n
  * Once information is sent correctly, it displays a QR code which encodes it, allowing the
user to scan it with a phone camera.
  * @{
  */

#include <Arduino.h>

#include <IRremoteESP8266.h>
#include <IRrecv.h>
#include <IRutils.h>
#include <IRsend.h>
#include <Adafruit_SSD1306.h>
#include <qrcode.h>

#include <EEPROM.h>

#include <CRC32.h>
#include <ESP8266TrueRandom.h>
#include <curve25519-donna.h>

#include <string.h>
#include <math.h>

// ****************************************************************

/**
 * \defgroup global_variables 1. Global Variables Definition
 * @{
 */

/**
 * \defgroup eeprom_control_variables 1.1. Variables to Control EEPROM Status and Content
 * @{
 */
/// Remote Controller ID - Always 1 (hardcoded).
const unsigned char devID = 1;

/// Node ID to Communicate to (Selected when pushing the button).
unsigned char selected_devID = 0;

/// Node ID to Communicate to. Auxiliar temporal var assigned in contactNode(), equal to
selected_devID unless the latter is 0.
unsigned char dest_devID;

/// EEPROM Control Char target, 'Y'.
const unsigned char CONTROL_EEPROM_TARGET_CHAR[2] = {'Y', '\0'};

/// EEPROM Read Control Char is stored here. If equal to 'Y', memory was written before.
unsigned char eeprom_control_char[2];

/// Max Number of Nodes is 206 (Limited by EEPROM).
#define NUM_NODES_MAX 206
```

```
/// Array of Nodes. +1 to store empty char '\0' at last position.
unsigned char nodes_array[NUM_NODES_MAX+1];
/// Number of Nodes. Default is 1. Value may change after reading EEPROM.
unsigned char number_nodes = 1;
/**@}*/

/**
 * \defgroup eeprom_address_map 1.2. EEPROM ADDRESS MAP
 * It uses 256 addresses for the Controller. Each address can store 8 bits.
 * @{
 */
/// 0x00, ssid size = 16 B.
#define ADDR_SSID 0
/// 0x10, psk size = 32 B.
#define ADDR_PSK 16
/// 0x30, n_nod size = 1 B.
#define ADDR_NUM_NODES 48
/// 0x31, number nodes up to 206 Nodes.
#define ADDR_NODE_1 49
/// 0xFF, control character. if char=='Y', memory has data.
#define ADDR_CONTROL_CHAR 255
/**@}*/

/**
 * \defgroup wifi_credentials 1.3. <SSID,PSK> global variables
 * @{
 */
/// SSID Max Size is 16 Bytes
#define SSID_MAX_SIZE 16
/// SSID Max Size is 32 Bytes
#define PSK_MAX_SIZE 32

/// Variables were EEPROM ssid data is stored. Initialized as empty "".
unsigned char ssid[SSID_MAX_SIZE+1] = "";
/// Variables were EEPROM psk data is stored. Initialized as empty "".
unsigned char psk[PSK_MAX_SIZE + 1] = "";
/**@}*/

/// Macro which translates a byte (8 bits) to 2 HEX Chars (4+4 bits).
#define TO_HEX(i) ( (i) <= 9 ? '0' + (i) : 'A' - 10 + (i) )

////////////////////////////////////////////////////////////////
// Variables used in IR Send/Receive
////////////////////////////////////////////////////////////////
/**
 * \defgroup ir_variables 1.4. Variables used in IR Send/Receive
 * @{
 */
/// IR RX Pin D6 GPIO12 in ESP8266 WifiKit 8.
const uint8_t kRecvPin = 12;
/// Set the GPIO to be used to receive data.
IRrecv irrecv(kRecvPin);

/// IR TX D7 GPIO13 in ESP8266 WifiKit 8.
const uint8_t kIrLed = 13;
/// Set the GPIO to be used to send data.
IRsend irsend(kIrLed);

/// Variable where a received message is stored to.
decode_results received_data;
/// Variable where a message is stored before being sent.
decode_results send_data;

/// Transmission and Reception Buffers Size.
#define MAX_PACKET_SEQUENCE 1024
/// Reception data buffer of 32bits/packet.
uint32_t rx_data_buffer[MAX_PACKET_SEQUENCE];
```

```
/// Transmission Data buffer of 32bits/packet.
uint32_t tx_data_buffer[MAX_PACKET_SEQUENCE];
/**@}*/

/////////////////////////////////////////////////////////////////
// Variables used in IR Frame Control
/////////////////////////////////////////////////////////////////
/** \defgroup frame_packet_types 1.5. Frame Packet Types
  * Variables used in IR Frame Control:
  * 3 bits for type of packet.\n
  * 8 bits for dest deviceID.\n
  * 32 bits for data.\n
  * 16 bits for CRC16 Checksum.\n
  * Type of Packet:\n
  * 0, 000 -> HELLO_ENCRYPTED.\n
  * 1, 001 -> HELLO.\n
  * 2, 010 -> HELLO_NEW.\n
  * 3, 011 -> DATA.\n
  * 4, 100 -> ACK.\n
  * 5, 101 -> DATA_END (Last Data Packet -> Checksum).\n
  * 6, 110 -> DATA_END_ENCRYPTED (Last Data Packet (encrypted) -> Checksum).\n
  * 7, 111 -> NACK - Not Used.\n
  * @{
  */
#define TYPE_HELLO_ENCRYPT 0b000
#define TYPE_HELLO 0b001
#define TYPE_HELLO_NEW 0b010
#define TYPE_DATA 0b011
#define TYPE_ACK 0b100
#define TYPE_DATA_END 0b101
#define TYPE_DATA_END_ENCRYPT 0b110
#define TYPE_NACK 0b111
/**@}*/

/**
 * \defgroup program_flags 1.6. FLAGS used by the program to distinguish between states
 * @{
 */
/// Hello Communication Flag. 1 means HELLO Exchange was done, so don't listen for more HELLO
Packets.
uint8_t FLAG_HELLO_DONE = 0;

/// Encryption Flag. 1 means Data is encrypted(DH Key Exchange). Default is 1.
uint8_t FLAG_ENCRYPT_DATA = 1;
/**@}*/


//////////////////////////////////////////
//  VARIABLES USED IN OLED DISPLAY
//////////////////////////////////////////
/**
 * \defgroup oled_display_variables 1.7. Variables used in OLED Display
 * @{
 */
/// OLED display width, 128 pixels.
#define SCREEN_WIDTH 128
/// OLED display height, 32 pixels.
#define SCREEN_HEIGHT 32

/// Displat Reset PIN Number, 16.
#define OLED_RESET 16

/// Init declaration for an SSD1306 display connected to I2C (SDA, SCL pins).
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RESET);

/// QR Code Variable to be printed in the display.
QRCode qrcode;
```

```
/**@}*/

/**
 * \defgroup button_interrupt_variables 1.8. Variables to handle Button Interruptions
 * @{
 */
/// PIN used by the input button of the micro.
#define BUTTON_PIN 0

/// Bool used to distinguish is button was pressed or released.
bool buttonIsHigh = false;
/// Timer representing the instant the button was pressed.
long timeInit = 0.0;
/// Timer representing the instant the button was released.
long timeEnd = 0.0;
/// Bool that indicates a menu needs to refresh to reflect changes due to user input
bool screen_needs_refresh = false;
/// Bool that turns true when a long buttom press is registered.
bool long_press = true;
/// Bool that allows program to exit devID selection loop.
bool devID_confirmed = true;
/// Bool that allows program to exit encryption selection loop.
bool encrypt_confirmed = true;
/// Bool that prints the stages of communication in the OLED display. It slows down the
transmission several ms.
bool verbose_display = false;
/// Unsigned char representing a number used to show progress in OLED display [x/10].
unsigned char current_entry = 1;
/**@}*/

/**@}*/

// ****************************************************************************

/**
 * \defgroup functions 2. Functions
 * Module with all the custom functions implemented in the controller source code
 * @{
 */

/**
 * \defgroup debug_functions 2.1. DEBUG Functions
 * Functions which are used for debugging purposes (printing logs and traces mainly)
 * @{
 */

/**
  *  DEBUG Function which prints in terminal the time elapsed in seconds.
  *   @param[in] long ref: Reference time where the counting period begins.
  */
void printTimeSeconds(long ref){
  double elapsed = (millis()-ref)/1000.0;
  Serial.printf(" - Time Elapsed: %.4f seconds.\n", elapsed);
}

/**
  * DEBUG function used to print in terminal information regarding received packets.
  *  @param[in] uint32_t header_capture: integer that represents the 32bits captured in a IR
frame.
  *  @param[in] uint16_t received_counter: integer that represents the current packet counter
in the data exchange.
  *  @param[in] uint8_t  received_type: integer that represents the type of packet that has
been received.
  */
void printCapturedSeq(uint32_t header_capture, uint16_t received_counter, uint8_t
received_type) {
  Serial.println("\nDEBUG: printCapturedSeq() HEX:");
```

```
  Serial.print("Captured: ");       Serial.println(header_capture, HEX);
  Serial.print("Counter: ");        Serial.println(received_counter, HEX);
  Serial.print("packetType: ");     Serial.println(received_type, HEX);
  Serial.println("*****\n");
}

/// DEBUG function used to print in terminal sent packet frame, represented in the global
variable send_data.
void printSentPacket() {
  Serial.println("printSentPacket() info:");
  serialPrintUint64(send_data.value, BIN);
  Serial.println("");
  serialPrintUint64(send_data.value, HEX);
  Serial.println("");
}

/// DEBUG function used to print in terminal received packet frame, represented in the global
variable received_data.
void printReceivedPacket() {
  Serial.println("printReceivedPacket() info:");
  //serialPrintUint64(received_data.value, BIN);
  //Serial.println("");
  serialPrintUint64(received_data.value, HEX);
  Serial.println("");
}

/**
  * DEBUG function used to print in terminal information regarding getRXFrameData() function
output.
   *  @param[in] uint8_t destID: integer that represents frame's Device Destination ID value.
   *  @param[in] uint8_t packet_type: integer that represents the current packet's type.
   *  @param[in] uint8_t is_from_controller: integer of which only the LS bit is used. 1 means
packet comes from controller device.
   *  @param[in] uint32_t payload: integer representing the payload data of the frame.
   */
void printRXPacketInfo(uint8_t destID, uint8_t packet_type, uint8_t is_from_controller,
uint32_t payload) {
  Serial.println("\nDEBUG: printRXPacketInfo()[BIN]: ");
  Serial.print("destID: ");             Serial.println(destID, BIN);
  Serial.print("packetType: ");         Serial.println(packet_type, BIN);
  Serial.print("isfromcontroller: ");   Serial.println(is_from_controller, BIN);
  Serial.print("payload: ");            Serial.println(payload, BIN);
  Serial.println("*****\n");
}

/// DEBUG function used to print stored EEPROM parameters: SSID, PSK, devID, number_nodes...
void printEEPROMData(){
  Serial.print("\nSSID: ");
  Serial.println((char *) ssid);
  Serial.print("PSK: ");
  for(int i=0; i<32; i++) {
    Serial.printf("%c", TO_HEX(psk[i]));
    }
    Serial.println();
  for(int i=0; i<32; i++) {
    Serial.printf("%d ", psk[i]);
  }Serial.println();
  Serial.print("DeviceID: ");
  Serial.println(devID);
  Serial.print("Number of Nodes: ");
  Serial.println(number_nodes);

  Serial.print("List of devIDs stored: ");
  for(int i=0; i<number_nodes; i++) {
    Serial.print(nodes_array[i]);
    Serial.print(", ");
  }
```

```
  Serial.println("");

  Serial.print("Control char. target/actual: ");
  Serial.print((char) CONTROL_EEPROM_TARGET_CHAR[0]);
  Serial.print("/");
  Serial.println((char) eeprom_control_char[0]);
  Serial.println("");
}
/**@}*/

/**
 * \defgroup aux_functions 2.2. Program Auxiliar Functions
 * Functions which are called to perform important processes during the execution of the main
functions
 * @{
 */

/**
  *  Function which delays execution a variable random amount of time.
  *   @param[in] bool short_time: If true, the delay will be a short delay.
  */
void wait(bool short_time) {
  if(short_time)
    delay(150+(rand()%10)); // Wait 150 to 160 ms
  else{delay(2000+(rand()%1000));} // wait 2000 to 3000 ms
}

/**
  * Function that generates a random SSID that will be saved into device EEPROM during its
initialization.
  *  @param[out] unsigned char* result: pointer to which random generated SSID will be saved.
  */
void generateRandomSSID(unsigned char* result) {
  const unsigned char seq[63] =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
  strncpy((char *) result, "IOT_", SSID_MAX_SIZE+1);

  for(int i=4; i < SSID_MAX_SIZE; i++){
    if      (i<10) { result[i] = seq[ESP8266TrueRandom.random(0,52)]; }
    else if (i==10){ result[i] = '-'; }
    else           { result[i] = seq[ESP8266TrueRandom.random(52, 63)]; }
  }
  Serial.print("New random SSID generated: ");
  Serial.println((char *) result);
}

/**
  * Function that generates a random PSK that will be saved into device EEPROM during its
initialization.
  *  @param[out] unsigned char* result: pointer to which random generated PSK will be saved.
  */
void generateRandomPSK(unsigned char* result) {
  for(int i=0; i < 4; i++){ // 1 long 32 bits, divided by 4bits/HEXchar = 8 chars -> need
array of 4 longs
    uint32_t aux = ESP8266TrueRandom.random();
    result[8*i+0] = (unsigned char) ((aux & 0xF0000000) >> 28);
    result[8*i+1] = (unsigned char) ((aux & 0x0F000000) >> 24);
    result[8*i+2] = (unsigned char) ((aux & 0x00F00000) >> 20);
    result[8*i+3] = (unsigned char) ((aux & 0x000F0000) >> 16);
    result[8*i+4] = (unsigned char) ((aux & 0x0000F000) >> 12);
    result[8*i+5] = (unsigned char) ((aux & 0x00000F00) >> 8 );
    result[8*i+6] = (unsigned char) ((aux & 0x000000F0) >> 4 );
    result[8*i+7] = (unsigned char) ((aux & 0x0000000F) >> 0 );

    // Serial.print("Raw 32bits: "); serialPrintUint64((uint64_t) aux, BIN);
    // Serial.print("\nChars from Byte: ");
    // Serial.printf("%c | %c | %c | %c | %c | %c | %c | %c\n\n", TO_HEX(result[8*i+0]),
```

```
TO_HEX(result[8*i+1]),  TO_HEX(result[8*i+2]),  TO_HEX(result[8*i+3]),
TO_HEX(result[8*i+4]),  TO_HEX(result[8*i+5]),  TO_HEX(result[8*i+6]),
TO_HEX(result[8*i+7]));
  }

  Serial.print("New random PSK generated: ");
  for(int i=0; i<32; i++){
    Serial.printf("%c", TO_HEX(result[i]));
  }
  Serial.println();
}

/**
  * Function used to simplify the process of reading a single entry stored in the EEPROM.
  *  @param[in] uint8_t start_addr: integer that represents the initial address where the
information is stored.
  *  @param[in] uint8_t num_bytes: integer that represents the number of bytes (or addresses)
to read.
  *  @param[out] unsigned char* result: pointer to where the read output should be stored.
  */
void readEEPROM(uint8_t start_addr, uint8_t num_bytes, unsigned char* result){
  EEPROM.begin(256);
  uint8_t index = 0;

  result[index] = (EEPROM.read(start_addr+index));
  while (index<num_bytes-1) {
    index = index+1;
    result[index] = (EEPROM.read(start_addr+index));
  }
  EEPROM.end();
}

/**
  * Function used to simplify the process of writing a single entry to the EEPROM.
  *  @param[in] uint8_t start_addr: integer that represents the initial address where the
information is stored.
  *  @param[in] unsigned char *data: pointer to where the read output should be stored.
  *  @param[in] uint8_t size: integer that represents the number of bytes (or addresses) to
read.
  */
void writeEEPROM(uint8_t start_addr, unsigned char *data, uint8_t size){
  EEPROM.begin(256);
  uint8_t index = 0;
  EEPROM.write(start_addr+index, data[index]);
  while(index < size-1){ //data[index] != '\0' &&
    index = index+1;
    EEPROM.write(start_addr+index, data[index]);
  }
  EEPROM.commit();
  EEPROM.end();
}

/// Function used to represent in the OLED display the device ID selection menu.
void displayDeviceIDSelection() {
  display.clearDisplay();
  display.display();
  display.setTextSize(1);
  display.setTextColor(WHITE);
  display.setCursor(0, 0);
  display.println(F("Select devID (Short Push) | Confirm (Long)"));
  display.print(F("["));
  display.print(current_entry);
  display.print(F("/"));
  display.print(number_nodes);

  if(selected_devID==0) {
    display.print(F("] Add New Device"));
```

```
  } else {
    display.print(F("] Connect to ID "));
    display.print(selected_devID);
  }
  display.display();
}

/// Function used to show in the display the Communication Encryption Selection Menu.
void displayEncryptedCommSelection() {
  display.clearDisplay();
  display.display();
  display.setTextSize(1);
  display.setTextColor(WHITE);
  display.setCursor(0, 0);
  display.println(F("Short Push: Switch | Long Push: Confirm"));
  display.print(F("Use Encrypted Communication?    "));
  if(FLAG_ENCRYPT_DATA) { display.print("YES"); Serial.println("Current Choice: YES"); }
  else                  { display.print("NO"); Serial.println("Current Choice: NO"); }
  display.display();
}

/// Function used to display QR Code in both OLED and Terminal
void displayOLEDQRCode() {
    display.clearDisplay();
    display.setTextSize(1);
    display.setTextColor(WHITE);
    display.setCursor(0, 0);
    display.display();
    wait(1);
    long dt = millis();
    uint8_t qrcodeData[qrcode_getBufferSize(3)]; // Version 3 -> size 29x29,  53 bytes of
data if low error correction
    char *str = (char *) malloc(53);
    char *psk_hex = (char *) malloc(PSK_MAX_SIZE+1);
    strcpy (str, (char*) ssid);
    strcat(str, ":\n");
    for(int i=0; i<PSK_MAX_SIZE; i++){
      psk_hex[i] = TO_HEX(psk[i]);
      Serial.printf("|%c|", TO_HEX(psk[i]));
    }
    strncat(str, psk_hex, 32);
    printf("\n%s\n", str);
    qrcode_initText(&qrcode, qrcodeData, 3, 0, str);
    printTimeSeconds(dt);

    for (uint8_t y = 0; y < 32; y++) {
      for (uint8_t x = 0; x < 128; x++) {
        display.drawPixel(x, y, WHITE);
      }
    }

    for (uint8_t y = 0; y < qrcode.size; y++) {
        Serial.print("        ");
        for (uint8_t x = 0; x < qrcode.size; x++) {
          // Print each module (UTF-8 \u2588 is a solid block)
          Serial.print(qrcode_getModule(&qrcode, x, y) ? "\u2588\u2588": "  ");
          if(qrcode_getModule(&qrcode, x, y)) {
            display.drawPixel(48+x, y+1, BLACK);
          }
        }
        Serial.println();
    }

    Serial.println("\n\n\n");
    free(str);
    free(psk_hex);
```

```
    display.ssd1306_command(SSD1306_SETCONTRAST);
    display.ssd1306_command(0);
    display.display();
    wait(0);
    wait(0);
    wait(0);
}

/// Function used to initialize the EEPROM with default parameters, generating random
<SSID,PSK>.
void setupEmptyEEPROM() {
  Serial.println("EEPROM WAS NOT SET UP. CHARACTER in 0xFF WAS NOT 'Y'. Proceeding to
initialize the memory with new SSID/PSK");
  generateRandomSSID(ssid);
  generateRandomPSK(psk);

  writeEEPROM(ADDR_SSID, ssid, SSID_MAX_SIZE);
  writeEEPROM(ADDR_PSK, psk, PSK_MAX_SIZE);

  // Conversion from int to char array - number of Nodes (initially 1)
  unsigned char number_nodes_char[1];
  number_nodes_char[0] = number_nodes;
  writeEEPROM(ADDR_NUM_NODES, number_nodes_char, 1);

  // Conversion from int to char array - devID of controller (1)
  unsigned char devID_char[1];
  devID_char[0] = devID;
  writeEEPROM(ADDR_NODE_1, devID_char, 1);

  // Write control char 'Y' in 0xFF to tell device's EEPROM has data
  strncpy((char *) &eeprom_control_char, (char *) &CONTROL_EEPROM_TARGET_CHAR, 2);
  writeEEPROM(ADDR_CONTROL_CHAR, eeprom_control_char, 1);
}

/// Function used to read all EEPROM configuration values and load them in the program global
variables.
void loadConfigValuesFromEEPROM() {
  // Read SSID,PSK
  readEEPROM(ADDR_SSID, SSID_MAX_SIZE, ssid);
  readEEPROM(ADDR_PSK, PSK_MAX_SIZE, psk);

  // Read Num of registered Nodes
  unsigned char number_nodes_char[2];
  readEEPROM(ADDR_NUM_NODES, 2, number_nodes_char);

  number_nodes = number_nodes_char[0];

  // Read each registered NodeID
  for(int i=0; i<number_nodes; i++){
    unsigned char current_devID[2];
    readEEPROM(ADDR_NODE_1+i, 2, current_devID);
    nodes_array[i] = current_devID[0];
    nodes_array[i+1] = '\0';
  }
}

/**
  * Function that performs the Data integrity Check for a burst of received packets.\n
  * It does a CRC32 check and determines if the data is valid.
  *  @param[in] uint16_t size: Number of received packets in the RX_buffer.
  *  @param[in] uint32_t rx_checksum: Checksum Received in the last data packet.
  * @return bool: True is data is valid. False if data is not valid.
  */
bool checkDataIntegrity(uint16_t size, uint32_t rx_checksum){
  uint32_t checksum = CRC32::calculate(rx_data_buffer, size);
  Serial.println("Checking Checksums...");
  Serial.printf("Received: %d | calculated: %d\n", rx_checksum, checksum);
```

```
  if(checksum == rx_checksum)
    return true;

  Serial.println("Checksums not matching!");
  return false;
}

/**
  * Function used to handle button interruptions, mainly used in selection menu navigation.\n
  * As there is a single button available, function is called in both press and release
actions.\n
  * It determines if the button was pressed or released, and differenciate between a short
and a long press.
  */
void handleButtonInterrupt() {
  if(!buttonIsHigh) { // Falling Edge (Low to High)
    long timeNow = millis();
    if(timeNow-timeInit<50) { //Anti-Bouncing
      timeInit = timeNow;
      return;
    }
    buttonIsHigh = true;
    Serial.println("BUTTON PRESSED");
    timeInit = millis();

    return;
  }

  // else -> Rising Edge (High to Low), button was released
  Serial.println("BUTTON RELEASED");
  buttonIsHigh = false;
  timeEnd = millis();
  Serial.print("Time Elapsed: ");
  Serial.println(timeEnd-timeInit);

  if(timeEnd-timeInit<=100) // Anti-Bouncing, too short press, do nothing
    return;

  if(timeEnd-timeInit<=1000) { // Short Press, change selected DeviceID
    timeEnd = 0.0;
    timeInit = 0.0;
    if(!devID_confirmed) {
      if(number_nodes>1) {
        Serial.println("Number nodes: ");
        Serial.println(number_nodes);
        if(selected_devID==nodes_array[number_nodes-1]){ // If Last Device was Selected, go
Back to 0
          selected_devID = 0;
          current_entry = 1;
          screen_needs_refresh = true;
          return;
        }
        if(selected_devID==0){ // Skip passing from new Device(0) to 1, as ID=1 is his own ID
          selected_devID = 2;
          current_entry = 2;
          screen_needs_refresh = true;
          return;
        }
        for(int id=0; id<number_nodes-1; id++) { // Iterate to go to the next ID in list
          if(selected_devID == nodes_array[id]) {
            selected_devID = nodes_array[id+1];
            current_entry = current_entry + 1;
            screen_needs_refresh = true;
            return;
          }
        }
      }
```

```
    } else if(!encrypt_confirmed) {
      FLAG_ENCRYPT_DATA = !FLAG_ENCRYPT_DATA;
      screen_needs_refresh = true;
      return;
    }
  } else { // Long Press (+1000ms), Confirm Selected Device
    long_press = true
    if(!devID_confirmed) { devID_confirmed = true; return; }
    else if(!encrypt_confirmed) { encrypt_confirmed = true; return; }
  }
}

/**
  * Function used to convert the info to send into a formatted frame ready to be sent.
  *   @param[in] uint8_t destID: device to which the packet is sent.
  *   @param[in] uint8_t packet_type: integer that represents the current packet's type.
  *   @param[in] uint32_t payload: integer representing the payload data of the frame.
  *  @return uint64_t: integer which represents the frame ready to be sent (up to 36 bits,
RC6 IR protocol).
  */
uint64_t buildTXFrame(uint8_t destID, uint8_t packet_type, uint32_t payload) {
  uint64_t data = 0ULL;
  if(packet_type==TYPE_ACK || packet_type==TYPE_NACK){
    data =
      static_cast<uint64_t>(payload & 0xFFF)     << 0  |
      static_cast<uint64_t>(1ULL)                << 12 | // Packet Sent From Controller,
always 1
      static_cast<uint64_t>(packet_type & 0b111) << 13;
  }
  else if(packet_type==TYPE_HELLO || packet_type==TYPE_HELLO_ENCRYPT ||
packet_type==TYPE_HELLO_NEW) {
    data =
      static_cast<uint64_t>(payload)     << 0  |
      static_cast<uint64_t>(1ULL)        << 24 | // Packet Sent From Controller, always 1
      static_cast<uint64_t>(destID)      << 25 |
      static_cast<uint64_t>(packet_type) << 33;
  }
  else if(packet_type==TYPE_DATA || packet_type==TYPE_DATA_END_ENCRYPT ||
packet_type==TYPE_DATA_END) {
    data =
      static_cast<uint64_t>(payload)     << 0  |
      static_cast<uint64_t>(1ULL)        << 32 | // Packet Sent From Controller, always 1
      static_cast<uint64_t>(packet_type) << 33;
  }

  //Serial.println("DEBUG: buildTXFRAME:");
    //serialPrintUint64(data, HEX);
    //Serial.println();
    //serialPrintUint64(data, BIN);
    //Serial.println("\n");
  return data;
}

/**
  * Function which processes the received frame stored in received_data global variable.\n
  * It processes the information of the frame and determines if the packet is valid.
  *   @param[in] uint16_t current_counter: represents the number of received packets in the
current burst.
  * @return uint32_t: integer which contains 2 different variables:\n
  *   Packet type (8bits)  in the 8 LSBs.\n
  *   New Counter (16bits) in the 16 MSBs. If NewCounter==current_counter_input+1, it means
the packet is valid. Everything else means packet was wrong and it get discarded.
  */
uint32_t getRXFrameData(uint16_t current_counter) {
  uint64_t data = received_data.value;
  uint16_t nbits = received_data.bits;
```

```
  uint8_t destID = 0;
  uint8_t packet_type = 0;
  uint32_t payload = 0;
  uint8_t is_from_controller = 1;

  if(nbits==16) { // Short Frame -> TYPE ACK, NACK, BYE
    packet_type = (uint8_t) ((data>>13) & 0b111); // 3 bits
    payload = (uint32_t) (data & 0xFFF); // 12 bits
    is_from_controller = (uint8_t) ((data>>12) & 0b1 ); // 1 bit
  }

  else if(nbits==36){ // Long Frame -> TYPE HELLO, DATA, DATA_END
    packet_type = (uint8_t) ((data>>33) & 0b111); // Located in same place for all 36-bit
frames

    if(packet_type==TYPE_HELLO || packet_type==TYPE_HELLO_ENCRYPT) {
      payload = (uint32_t) (data & 0x00FFFFFF); // 24 bits
      is_from_controller = (uint8_t) ((data>>24) & 0b1 );
      destID = (uint8_t) (data>>25); //DestID only present in HELLO seqs
    }
    else if(packet_type==TYPE_HELLO_NEW) {
      payload = (uint32_t) (data & 0x00FFFFFF); // 24 bits
      is_from_controller = (uint8_t) ((data>>24) & 0b1 );
      destID = (uint8_t) (data>>25); //DestID only present in HELLO seqs
    }
    else if (packet_type==TYPE_DATA || packet_type==TYPE_DATA_END ||
packet_type==TYPE_DATA_END_ENCRYPT){
      payload = (uint32_t) (data & 0xFFFFFFFF); // 32 bits
      is_from_controller = (uint8_t) ((data>>32) & 0b1 );
    }
  }

  if(is_from_controller) {Serial.println("\n**a** (not coming from a node)");return -1;}

  uint16_t next_counter = current_counter + 1;
  uint32_t response_ok = packet_type | (((uint32_t) (next_counter))<<16);
  uint32_t response_ko = packet_type;

  //Serial.print("DEBUG: responseOK: ");
  //serialPrintUint64(response_ok, BIN);
  //Serial.println("\n\n");

  switch (packet_type) {
    case TYPE_ACK:
      //Serial.println("*Got ACK*");
        //Serial.printf("Payload: %d, current_counter: %d. Are equal: ", payload,
current_counter);
        //Serial.println((uint16_t) (payload)==current_counter);
      if ((uint16_t) (payload)==current_counter) { //ACKs should contain nº of packets
received
        Serial.printf("Got ACK number %d\n", current_counter);
        //printRXPacketInfo(destID, packet_type, is_from_controller, payload);
        return response_ok;
      }
      return response_ko;
      break;

    case TYPE_NACK:
      return response_ko;
      break;

    case TYPE_HELLO_ENCRYPT:
      if(FLAG_HELLO_DONE)
        return response_ko;
      if(devID!=destID)
        return response_ko;
```

```
      Serial.println("Got Hello Encrypt!");
      //printRXPacketInfo(destID, packet_type, is_from_controller, payload);
      FLAG_ENCRYPT_DATA = 1;
      return response_ok;
      break;

    case TYPE_HELLO:
      if(FLAG_HELLO_DONE)
        return response_ko;
      if(devID!=destID)
        return response_ko;

      Serial.println("Got Hello!");
      //printRXPacketInfo(destID, packet_type, is_from_controller, payload);
      FLAG_ENCRYPT_DATA = 0;
      return response_ok;
      break;

    case TYPE_HELLO_NEW:
      if(FLAG_HELLO_DONE)
        return response_ko;
      if(payload==0)
        return response_ko;

      Serial.println("Got Hello_NEW!");
      if(destID==devID)
        return response_ok;

      return response_ko;
      break;

    case TYPE_DATA:
      if(current_counter>0 && rx_data_buffer[current_counter-1]==payload && payload>0)
        return response_ko;

      //Serial.printf("Got Data! counter: %d\n", current_counter);
      //printRXPacketInfo(destID, packet_type, is_from_controller, payload);
      rx_data_buffer[current_counter] = payload;
      return response_ok;
      break;

    case TYPE_DATA_END:
      if(current_counter==0)
        return response_ko;

      //Serial.printf("Got *LAST* Data! counter: %d\n", current_counter);
      if(checkDataIntegrity(current_counter, payload)) {
        FLAG_ENCRYPT_DATA = 0;
        return response_ok;
      }
      return response_ko;
      break;

    case TYPE_DATA_END_ENCRYPT:
      if(current_counter==0)
        return response_ko;

      //Serial.printf("Got *LAST* Data! counter: %d\n", current_counter);
      if(checkDataIntegrity(current_counter, payload)) {
        FLAG_ENCRYPT_DATA = 1;
        return response_ok;
      }
      return response_ko;
      break;

    default:
      return response_ko;
```

```
      break;
    }
}

/**
  * Function which encrypts the data before sending it over IR. It uses a 32-byte shared_key
to encrypt 32 bytes of information.\n
  *  As the key is unique in each session, there is no need to use a Init_Vector, as in this
case:\n
  *  len(data)==len(key), we can use a simple XOR bit comparison to encrypt/decrypt the Data.
  *   @param[in] byte *shared_key: Pointer to the session key used to encrypt/decrypt the
data.
  *   @param[in] byte *msg: Pointer to the message to be encrypted.
  *   @param[in] bool hasPubKey: Boolean which, if true, allow to append the publickey in
clear to the message. Default false.
  *   @param[out] byte *encrypted: Pointer to where the encrypted result will be saved.
 */
void encryptData(byte *shared_key, byte *msg, byte *encrypted, bool hasPubKey = false) {
  for(int i=0; i<SSID_MAX_SIZE; i++){
    encrypted[i] = msg[i]^shared_key[i];
    encrypted[i+SSID_MAX_SIZE] = msg[i+SSID_MAX_SIZE]^shared_key[i+SSID_MAX_SIZE];
    if(hasPubKey) {
      encrypted[i+PSK_MAX_SIZE] = msg[i+PSK_MAX_SIZE];
      encrypted[i+PSK_MAX_SIZE+SSID_MAX_SIZE] = msg[i+PSK_MAX_SIZE+SSID_MAX_SIZE];
    }
  }
}

/**
  * Function which decrypts the data received over IR. It uses a 32-byte shared_key to
encrypt 32 bytes of information.\n
  * As the key is unique in each session, there is no need to use a Init_Vector, as in this
case:\n
  * len(data)==len(key), we can use a simple XOR bit comparison to encrypt/decrypt the Data.
  *   @param[in] byte *shared_key: Pointer to the session key used to encrypt/decrypt the
data.
  *   @param[in] byte *encrypted: Pointer to the message to be decrypted.
  *   @param[out] byte *decrypted: Pointer to where the decrypted result will be saved.
 */
void decryptData(byte *shared_key, byte *encrypted, byte *decrypted) {
  for(int i=0; i<SSID_MAX_SIZE; i++){
    decrypted[i] = encrypted[i]^shared_key[i];
    decrypted[i+SSID_MAX_SIZE] = encrypted[i+SSID_MAX_SIZE]^shared_key[i+SSID_MAX_SIZE];
  }
}

/**
  * Function that translates data char array ("ssid_1234") (Between 16 and 32 bytes) to
chunks of 32bits.
  *   @param[in] byte *char_data: Pointer to the data 8-bits array to be converted into a
24bits array.
  *   @param[in] int size: size of the data in bytes.
  *   @param[in] bool is_psk: Optimizes packet building in case the data is from the PSK, as
it uses only 4 bits for each character (50% efficiency otherwise). Default is false.
  * @return uint16_t: integer representing the number of data packets the TX_buffer contains.
  */
uint16_t buildTXDataBuffer(byte *char_data, int size, bool is_psk=false) {
  uint16_t number_chunks = (size*8)/32;
  uint32_t checksum;

  // Casting byte int array into uint32 array.
  // 1 uint32 will hold 4 bytes, as payload is 32bits.
  Serial.println("DEBUG: buildTXDataBuffer [HEX]:");

  if(is_psk){
    for(int i=0; i<32; i++){
      Serial.printf("%c", TO_HEX(char_data[i]));
```

```
    }Serial.println();
    number_chunks = number_chunks/2;
    for(int i=0; i<number_chunks; i++) {
      tx_data_buffer[i] =
        static_cast<uint32_t> (char_data[8*i])   << 0 |
        static_cast<uint32_t> (char_data[8*i+1]) << 4 |
        static_cast<uint32_t> (char_data[8*i+2]) << 8 |
        static_cast<uint32_t> (char_data[8*i+3]) << 12|
        static_cast<uint32_t> (char_data[8*i+4]) << 16|
        static_cast<uint32_t> (char_data[8*i+5]) << 20|
        static_cast<uint32_t> (char_data[8*i+6]) << 24|
        static_cast<uint32_t> (char_data[8*i+7]) << 28;
      serialPrintUint64(tx_data_buffer[i], HEX);
      Serial.println();
    }

  }
  else {
    for(int i=0; i<number_chunks; i++) {
        tx_data_buffer[i] =
          static_cast<uint32_t> (char_data[4*i])   << 0 |
          static_cast<uint32_t> (char_data[4*i+1]) << 8 |
          static_cast<uint32_t> (char_data[4*i+2]) << 16|
          static_cast<uint32_t> (char_data[4*i+3]) << 24;
        //serialPrintUint64(tx_data_buffer[i], HEX);
        //Serial.println();
    }
  }

  // Avoid sending many 0's packet at the end of data.
  for(int i=0; i<number_chunks; i+=0){
    if(tx_data_buffer[i]==0) {
      for (int j=i; j<number_chunks-1; j++) {
        tx_data_buffer[j] = tx_data_buffer[j+1];
      }
      number_chunks -= 1;
    } else{i+=1;}
  }

  // Checksum calculation
  checksum = CRC32::calculate(tx_data_buffer, number_chunks);
  tx_data_buffer[number_chunks] = checksum;
  number_chunks += 1;

  Serial.printf("Num of data packets: %d\n\n", number_chunks);
  return number_chunks;
}

/**
  * Function that populates the transmission buffer with the PSK and SSID to be sent.\n
  * It also calculates the checksum of the sending data.
  *   @param[in] byte *shared_key: Pointer to the session key shared with the addressee. If
not provided, data will not be encrypted. Default is NULL
  *   @param[in] byte *mypublic_key: Pointer to the device public key that will be
transmitted. If not provided, burst will not include those extra 32 bytes. Default is NULL
  * @return uint16_t: integer representing the number of data packets the TX_buffer contains.
  */
uint16_t buildAllDataIntoBuffer(byte *shared_key=NULL, byte *mypublic_key=NULL) {
  uint16_t num_bytes; // 1 PSK char is only 4 bits, we can get 2 chars in a Byte

  if(mypublic_key) { num_bytes = SSID_MAX_SIZE+PSK_MAX_SIZE/2+32; } // 32 B is pubkey length
  else             { num_bytes = SSID_MAX_SIZE+PSK_MAX_SIZE/2; }

  uint16_t total_packets = num_bytes/4; //4 bytes per packet
  uint32_t checksum;
  byte msg[num_bytes];
```

```
  Serial.println("DEBUG: buildAllDataIntoBuffer():");

  for(int i=0; i<SSID_MAX_SIZE; i++) { // msg[0], msg[1], ..., msg[31] = ssid[0], ssid[1],
..., psk[30]+psk[31]
    msg[i] = ssid[i];

    msg[i+16] = psk[2*i+0]<<0 | (psk[2*i+1]<<4);

    if(mypublic_key) { // Loop goes from 0 to 15 only, we need to access 32 bytes in position
0-31 and allocate them in position 32-63
      msg[i+32] = mypublic_key[i];
      msg[i+48] = mypublic_key[i+16];
    }
  }

  if(shared_key) {

    //Serial.println("MSG before encryption: ");
    //for(int i=0; i<num_bytes; i++) {
    //  Serial.printf("Message i: %d = %d\n", i, msg[i]);
    //}
    if(mypublic_key) {
      encryptData(shared_key, msg, msg, true); //result is returned into msg variable
    } else {
      encryptData(shared_key, msg, msg, false);
    }

    //Serial.println("MSG after encryption: ");
    //for(int i=0; i<num_bytes; i++) {
    //  Serial.printf("EncrMessage i: %d = %d\n", i, msg[i]);
    //}
  }

  for(int i=0; i<total_packets; i++) {
    tx_data_buffer[i] =
      static_cast<uint32_t> (msg[4*i])          |
      static_cast<uint32_t> (msg[4*i+1]) << 8  |
      static_cast<uint32_t> (msg[4*i+2]) << 16|
      static_cast<uint32_t> (msg[4*i+3]) << 24;
    serialPrintUint64(tx_data_buffer[i], HEX);
    Serial.println();
  }

  // Checksum calculation
  checksum = CRC32::calculate(tx_data_buffer, total_packets);
  tx_data_buffer[total_packets] = checksum;
  total_packets += 1;

  Serial.printf("Num of data packets: %d\n\n", total_packets);
  return total_packets;
}

/**
  * Core Function which establishes a connection with a Node, transfers the Wi-Fi credentials
and closes the connection.\n
  * It is called periodically, until receiving a confirmation with Destination Node that info
was correctly received.\n
  * It includes several connection stages:\n
  *     HELLO_NEW: Handshake with a random proof for new devices, which allows concurrence
problems with multiple new devices answering the controller at the same time.\n
  *     HELLO: Normal handshake, performed for all devices. it also informs the node if the
transmission will be encrypted or not.\n
  *     KEY_EXCHANGE: If users selects Encryption, there is a key exchange stage where both
nodes send each other their public keys.\n
  *     DATA_EXCHANGE: Sends the data (in clear or encrypted) until receiving final ACK from
node.
  * @return bool: True if data could be transmitted. False otherwise.
```

```
  */
bool contactWithNode() {
  FLAG_HELLO_DONE = 0;

  long ref_time = millis();
  long time0 = millis();
  bool HELLO_PENDING = true;
  bool NEW_HELLO_PENDING = true;

  uint32_t rand_seq_hello = 0;

  // Send Broadcast (dest_addr=0, TYPE_HELLO_NEW, new ID in payload)
  if(selected_devID==0) {
    dest_devID = number_nodes+1;
    uint8_t counter = 1;
    send_data.value = buildTXFrame(0, TYPE_HELLO_NEW, dest_devID);
    irsend.sendRC6(send_data.value, 36);
    irrecv.resume();

    Serial.print("*TIME: Broadcast NEW HELLO Sent! ");
    printTimeSeconds(ref_time);

    //Wait for Node HELLO Answer with Random Payload
    while(NEW_HELLO_PENDING) {
      delay(71);
      if(irrecv.decode(&received_data)) {
        if(received_data.decode_type==RC6) {
          uint32_t header_capture = getRXFrameData(0);
          uint8_t received_type = (uint8_t) (header_capture);
          uint16_t received_counter = (uint16_t) (header_capture>>16);
          if(received_type==TYPE_HELLO_NEW && received_counter==1) {
            uint32_t payload = (uint32_t) (received_data.value & 0x00FFFFFF); // 24 bits;
              NEW_HELLO_PENDING = false;
              rand_seq_hello = payload;
              Serial.printf("Received Hello with a Random Payload: %d\n", rand_seq_hello);
          }
        }
        if(NEW_HELLO_PENDING && counter%3!=0)
          irrecv.resume();
      }
      if(NEW_HELLO_PENDING && counter%3==0) {
        delay(33);
        irsend.sendRC6(send_data.value, 36);
        irrecv.resume();
        Serial.println("\nSent new Broadcast HELLO!");
      }

      counter++;
    }
    Serial.print("*TIME: Broadcast NEW HELLO Answer Received! ");
    printTimeSeconds(ref_time);
    printTimeSeconds(time0);

    if(verbose_display) {
      display.clearDisplay();
      display.display();
      display.setTextSize(1);
      display.setTextColor(WHITE);
      display.setCursor(0, 0);
      display.println(F("Communication Established"));
      display.println(F("New Hello Answer Received"));
      display.display();
    }
  }
  else { // selected_devID>0 (existing device)
    dest_devID = selected_devID;
    Serial.println(dest_devID);
```

```
    rand_seq_hello = selected_devID;
  }

  // *** HELLO FOR A KNOWN DEVICE STARTS HERE ***

  // In case dest_devID was not known, send back hello with randomGen payload (rand_seq=0 if
dev is known)
  if(FLAG_ENCRYPT_DATA) { send_data.value = buildTXFrame(dest_devID, TYPE_HELLO_ENCRYPT,
rand_seq_hello); }
  else { send_data.value = buildTXFrame(dest_devID, TYPE_HELLO, rand_seq_hello); }

  irsend.sendRC6(send_data.value, 36);
  irrecv.resume();

  Serial.print("*TIME: HELLO for Known DevID SENT! ");
  printTimeSeconds(ref_time);

  time0 = millis();
  uint8_t counter = 1;

  //Wait for Node HELLO Answer
  while(HELLO_PENDING) {
    delay(71);
    if(irrecv.decode(&received_data)) {
      if(received_data.decode_type==RC6) {
        uint32_t header_capture = getRXFrameData(0);
        uint8_t received_type = (uint8_t) (header_capture);
        uint16_t received_counter = (uint16_t) (header_capture>>16);
        if( (received_type==TYPE_HELLO || received_type==TYPE_HELLO_ENCRYPT) &&
received_counter==1) {
          uint8_t payload = (uint8_t) received_data.value;
          if (payload == dest_devID) {
            HELLO_PENDING = false;
            Serial.println("Received Hello for Known DevID Answer!");
          }
        }
      }
      if(HELLO_PENDING && counter%3!=0)
        irrecv.resume();
    }

    if(HELLO_PENDING && counter%3==0) {
        delay(33);
        irsend.sendRC6(send_data.value, 36);
        irrecv.resume();
        Serial.println("\n Sent another HELLO for Known DevID!");
    }
    counter++;

    if(HELLO_PENDING && millis()-time0 > 5000)
      return false;
  }

  if(!FLAG_ENCRYPT_DATA) {
    Serial.print("*TIME: HELLO Answer Received! ");
    printTimeSeconds(ref_time);
    printTimeSeconds(time0);

    if(millis()-time0<600)
      delay(610-(millis()-time0)); // Wait for receiver to stop sending hellos, before
sending us
  }

  FLAG_HELLO_DONE = 1; // Flag=1 means we are ready to store data packets (in case they
arrive)
  Serial.print("*TIME: HELLO FINAL answer received! ");
  printTimeSeconds(ref_time);
```

```
  printTimeSeconds(time0);

  if(verbose_display) {
    display.clearDisplay();
    display.display();
    display.setTextSize(1);
    display.setTextColor(WHITE);
    display.setCursor(0, 0);
    display.println(F("Hello Final Answer Received"));
    if(FLAG_ENCRYPT_DATA)
      display.println(F("Sending PublicKey for Encryption..."));
    else { display.println(F("Sending Data in Clear...")); }
    display.display();
  }

  uint8_t packet_max_number;


  // **************** HELLO DONE, Exchanging Curve PublicKeys ****************



  if(FLAG_ENCRYPT_DATA) {
    static const uint8_t basepoint[32] = {9};
    uint8_t mycurve_secret[32];
    uint8_t mycurve_public[32];
    uint8_t hiscurve_public[32];
    uint8_t mycurve_shared[32];

    for(int i=0; i<32; i++) { mycurve_secret[i] = ESP8266TrueRandom.randomByte(); }
    curve25519_donna(mycurve_public, mycurve_secret, basepoint);
    //for(int i=0; i<32; i++) { Serial.printf("Mycurve_public Byte %d: %d\n", i,
mycurve_public[i]); }

    // *** Waiting for Device's PublicKey Reception ***


    bool IS_LAST_DATA = false;
    bool CHECKSUM_ERROR;
    time0 = millis();
    uint16_t packet_sequence;

    // Loop for Device's PublicKey Reception
    while(!IS_LAST_DATA) {
      CHECKSUM_ERROR = false;
      packet_sequence = 0;
      irrecv.resume();

      if(millis()-time0>1500)
        return false;

      while(!IS_LAST_DATA && !CHECKSUM_ERROR) {
        delay(2);
        if(irrecv.decode(&received_data)) {
          if(received_data.decode_type == RC6) {
            uint32_t header_capture = getRXFrameData(packet_sequence);
            uint8_t received_type = (uint8_t) (header_capture);
            uint16_t received_counter = (uint16_t) (header_capture>>16);

            if( received_type==TYPE_DATA && received_counter==packet_sequence+1 ) {
              Serial.printf("PublicKey Data %d received.\n", packet_sequence);
              packet_sequence++;
            }
            if(received_type==TYPE_DATA_END_ENCRYPT) {
              if(received_counter==packet_sequence+1){
                Serial.printf("**LAST** PublicKey Packet %d received.\n", packet_sequence);
                IS_LAST_DATA = true;
```

```
                packet_sequence++;
              } else { Serial.println("Checksum ERROR"); memset(&rx_data_buffer, 0,
packet_sequence); CHECKSUM_ERROR = true; }
            }
          }
          if(!IS_LAST_DATA)
            irrecv.resume();
        }
        if(packet_sequence>=9)
          CHECKSUM_ERROR=true;
      }
    }

    Serial.print("*TIME: Device's PublicKey Received");
    printTimeSeconds(ref_time);
    printTimeSeconds(time0);

    Serial.println("Sending DATA ACK");
    send_data.value = buildTXFrame(selected_devID, TYPE_ACK, packet_sequence);
    time0 = millis();

    // ACK for device's PublicKey
    for(int i=0; i<4; i++) {
      delay(5);
      irsend.sendRC6(send_data.value, 16); //ACK
    }
    //while(millis()-time0 < 180) {
    //  irsend.sendRC6(send_data.value, 16);
    //  delay(15);
    //}

    Serial.print("*TIME: ACK for PublicKey Sent - Pre-Saving Data");
    printTimeSeconds(ref_time);
    printTimeSeconds(time0);

    for(int i=0; i<packet_sequence-1; i++) { // PublicKey is 32bytes, there are 4bytes per
packet: packet_seq=8
      hiscurve_public[4*i+0] = (uint8_t) (rx_data_buffer[i] >> 0*8);
      hiscurve_public[4*i+1] = (uint8_t) (rx_data_buffer[i] >> 1*8);
      hiscurve_public[4*i+2] = (uint8_t) (rx_data_buffer[i] >> 2*8);
      hiscurve_public[4*i+3] = (uint8_t) (rx_data_buffer[i] >> 3*8);
      //Serial.printf("Buffer HisPublicKey %d: %d | %d | %d | %d\n", i,
hiscurve_public[4*i+0], hiscurve_public[4*i+1], hiscurve_public[4*i+2],
hiscurve_public[4*i+3]);
      //Serial.println();
    }
    memset(&rx_data_buffer, 0, packet_sequence);

    curve25519_donna(mycurve_shared, mycurve_secret, hiscurve_public);
    //for(int i=0; i<32; i++) { Serial.printf("Mycurve_shared Byte %d: %d\n", i,
mycurve_shared[i]); }

    Serial.println("Entering Encrypting function...");
    packet_max_number = buildAllDataIntoBuffer(mycurve_shared, mycurve_public);

    if(verbose_display) {
      display.clearDisplay();
      display.display();
      display.setTextSize(1);
      display.setTextColor(WHITE);
      display.setCursor(0, 0);
      display.println(F("PublicKey Received & SharedSecret calculated!"));
      display.println(F("Sending Encrypted Data..."));
      display.display();
    }
  }
  else { packet_max_number = buildAllDataIntoBuffer(); }
```

```
// **************** Curve Public Key Exchange Done, Data TX Starts Here ****************


bool ACK_PENDING = true;
time0 = millis();

while(ACK_PENDING) {
  Serial.println("\nEntering Sending DATA Loop...");
  for(int i=0; i<packet_max_number; i++){
    delay(5);

    if(millis()-time0 >2500) {
      break;
    }

    if(i+1==packet_max_number){
      if(FLAG_ENCRYPT_DATA){ send_data.value = buildTXFrame(selected_devID,
TYPE_DATA_END_ENCRYPT, tx_data_buffer[i]); }
      else {                send_data.value = buildTXFrame(selected_devID, TYPE_DATA_END,
tx_data_buffer[i]); }
    }
    else {                  send_data.value = buildTXFrame(selected_devID, TYPE_DATA,
tx_data_buffer[i]); }

    Serial.printf("Packet number %d\n", i);
    irsend.sendRC6(send_data.value, 36);
  }

  Serial.print("*TIME: All info sent");
  printTimeSeconds(ref_time);
  printTimeSeconds(time0);

  irrecv.resume();
  Serial.println("Waiting for ACK");
  time0 = millis();
  uint8_t n_attempts = 0; // Return after 3 missing acks

  while(ACK_PENDING) {
    delay(2);
    if(irrecv.decode(&received_data)) {
      if(received_data.decode_type == RC6) {
        uint32_t header_capture = getRXFrameData(packet_max_number);
        uint8_t received_type = (uint8_t) (header_capture);
        uint16_t received_counter = (uint16_t) (header_capture>>16);

        if( received_type==TYPE_ACK && received_counter==packet_max_number+1 ) {
          ACK_PENDING = false;
          Serial.println("\n\n**Burst ACK Received for all DATA**\n");
        }
      }
      if(ACK_PENDING)
        irrecv.resume();
    }
    if(millis()-time0 > 300)
      break;
  }

  if(ACK_PENDING){
    n_attempts += 1;
    if(n_attempts>=3)
      return false;
  }
}
```

```
  Serial.print("*TIME: DATA ACK received ");
  printTimeSeconds(ref_time);
  printTimeSeconds(time0);

  if(verbose_display) {
    display.clearDisplay();
    display.display();
    display.setTextSize(1);
    display.setTextColor(WHITE);
    display.setCursor(0, 0);
    display.println(F("Data Correctly Sent!"));
    display.println(F("ACK Final Received. Printing QRCode..."));
    display.display();
  }
  wait(0);

  // QR code Printing
  long_press = false;
  while(!long_press) {
    displayOLEDQRCode();
  }

  if(selected_devID==0) {
    Serial.println("Saving New NodeID into Memory:");

    number_nodes += 1;
    unsigned char array_number_nodes[1];
    array_number_nodes[0] = number_nodes;
    writeEEPROM(ADDR_NUM_NODES, array_number_nodes, 1);

    unsigned char array_dest_devID[1];
    array_dest_devID[0] = dest_devID;
    writeEEPROM(ADDR_NODE_1+number_nodes-1, array_dest_devID, 1);
  }

  return true;
}
/**@}*/

/**
 * \defgroup main_functions 2.3. Program Main Functions
 * Functions which must be implemented by an ESP8266. They execute the main logic of the
program with the aid of the previous functions
 * @{
 */

/// Setup function which checks EEPROM, generate random <SSID,PSK> if not initialized, waits
for user input in destID selection and Encryption menus, communicates with node, and prints
the QR_Code.
void setup() {
  Serial.begin(115200);
  while (!Serial)
    delay(50);
  Serial.println();

  time_t t;
  srand((unsigned) time(&t));

  pinMode(BUTTON_PIN, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(BUTTON_PIN), handleButtonInterrupt, CHANGE);

  //Uncomment this to reset memory by removing 'Y' control char
  /* unsigned char a[2] = {'A', '\0'};
   * writeEEPROM(ADDR_CONTROL_CHAR, a, 1);
   */
```

```
  readEEPROM(ADDR_CONTROL_CHAR, 2, eeprom_control_char);
  Serial.println("Memory before loading EEPROM:");
  printEEPROMData();

  // Control Char not found, init EEPROM (random <ssid,psk>, num_nodes=1, devID_1=1,
control_char=1)
  if ( !(eeprom_control_char[0]==CONTROL_EEPROM_TARGET_CHAR[0]) )
    setupEmptyEEPROM();

  loadConfigValuesFromEEPROM();
  Serial.println("Memory AFTER loading EEPROM:");
  printEEPROMData();

  // ***********************************************

  Serial.println("Select devID (ShortPush)\n    &\nConfirm (Long Push)\n");
  // number_nodes = 6;
  // unsigned char aux_array[6] = {1, 2, 3, 4, 5, 6};
  // strncpy((char *) &nodes_array, (char *) &aux_array, number_nodes);

  for(int i=0; i<number_nodes; i++) {
    Serial.print(nodes_array[i]);
    Serial.print(", ");
  }
  Serial.println();

  // SSD1306_SWITCHCAPVCC = generate display voltage from 3.3V internally
  if(!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) { // Address 0x3C for 128x32
    Serial.println(F("SSD1306 allocation failed"));
  }

  // QR code Printing
  long_press = false;
  while(!long_press) {
    displayOLEDQRCode();
  }

  display.display();
  displayDeviceIDSelection();
  devID_confirmed = true;

  while(!devID_confirmed) {
    if(screen_needs_refresh){
      screen_needs_refresh = false;
      displayDeviceIDSelection();
    }
    delay(100);
  }

  Serial.println("Want to Encrypt Transmission?");
  displayEncryptedCommSelection();
  encrypt_confirmed = true;

  while(!encrypt_confirmed) {
    if(screen_needs_refresh){
      screen_needs_refresh = false;
      displayEncryptedCommSelection();
    }
    delay(100);
  }

  display.clearDisplay();
  display.display();
  display.setTextSize(1);
  display.setTextColor(WHITE);
  display.setCursor(0, 0);
  display.println(F("Selection Saved!"));
```

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

telecom BCN

```
  display.print(F("Device Selected: "));
  if(selected_devID==0){
    display.println(F("NEW."));
  } else {
    display.print(selected_devID);
    display.println(F("."));
  }
  display.println(F("Trying to Communicate"));
  display.println(F("Point towards it..."));
  display.display();

  // ********************************************

  irrecv.enableIRIn();  // Getting IR Receiver Ready
  irsend.begin();       // Getting IR Transmitter Ready

  int num_attempts = 1;
  Serial.print("Contacting with the node. Current Attempt: ");
  Serial.println(num_attempts);

  while(!contactWithNode()){
    num_attempts += 1;
    Serial.print("Contacting with the node. Current Attempt: ");
    Serial.println(num_attempts);
  }

  Serial.printf("\nInfo Correctly Sent to Node ID: %d!\n", dest_devID);
}


/// Loop() function not used. Just sleep.
void loop() {
  delay(50);
}
/**@}*/

/**@}*/
/**@}*/
```

Annex II - ESP8266 Receiver Code

```
/**
  * \defgroup main main.cpp TFM Receiver Code
  * 2019 - Rafael de la Rosa Ortiz \n
  * Allows a ESP8266 Node to receive a <SSID,PK> from an IoT Controller.\n
  * It checks if device has EEPROM initialized. If not, it will start listening for a
controller until receiving <SSID,PK>\n
  * It receives the Key from the controller securely by implementing a Diffie Hellman Key
Exchange Protocol.\n
  * Once information is received correctly, it saves it in its EEPROM and tries to connect to
the Access Point with the credentials.
  * @{
  */

#include <Arduino.h>

#include <IRremoteESP8266.h>
#include <IRrecv.h>
#include <IRutils.h>
#include <IRsend.h>

#include <EEPROM.h>

#include <CRC32.h>
#include <ESP8266TrueRandom.h>
#include <curve25519-donna.h>

#include <string.h>
#include <math.h>

#include <ESP8266WiFi.h>
#include <WiFiClient.h>

// ****************************************************************

/**
 * \defgroup global_variables 1. Global Variables Definition
 * @{
 */

/**
 * \defgroup eeprom_control_variables 1.1. Variables to Control EEPROM Status and Content
 * @{
 */

/// Node Device ID - If 0, device has no ID assigned (it will start listening for a
controller).
unsigned char devID = 0;
/// Auxiliar device ID. It is temporal during the communication. Will be copied to devID if
communication ends successfully.
unsigned char devID_aux = 0;

/// Controller Device ID=1 (hardcoded). Nodes can only communicate with the controller.
const unsigned char CONTROLLER_ID = 1;

/// EEPROM Control Char target, 'Y'.
const unsigned char CONTROL_EEPROM_TARGET_CHAR[2] = {'Y', '\0'};

/// EEPROM Read Control Char is stored here. If equal to 'Y', memory was written before.
unsigned char eeprom_control_char[2];

/// Max Number of Nodes is 206 (Limited by EEPROM).
#define NUM_NODES_MAX 206
/// Array of Nodes. +1 to store empty char '\0' at last position.
unsigned char nodes_array[NUM_NODES_MAX+1];
```

```
/// Number of Nodes. Default is 1. Value may change after reading EEPROM.
unsigned char number_nodes = 1;
/**@}*/

/**
 * \defgroup eeprom_address_map 1.2. EEPROM ADDRESS MAP
 * It uses 64 addresses for an IoT node. Each address can store 8 bits.
 * @{
 */
/// 0x00, ssid size = 16 B.
#define ADDR_SSID 0
/// 0x10, psk size = 32 B.
#define ADDR_PSK 16
/// 0x30, n_nod size = 1 B.
#define ADDR_DEV_ID 48
/// 0x3F, control character. if char=='Y', memory has data.
#define ADDR_CONTROL_CHAR 63
/**@}*/

/**
 * \defgroup wifi_credentials 1.3. <SSID,PSK> global variables
 * @{
 */
/// SSID Max Size is 16 Bytes
#define SSID_MAX_SIZE 16
/// SSID Max Size is 32 Bytes
#define PSK_MAX_SIZE 32

/// Variables were EEPROM ssid data is stored. Initialized as empty "".
unsigned char ssid[SSID_MAX_SIZE+1] = "";
/// Variables were EEPROM psk data is stored. Initialized as empty "".
unsigned char psk[PSK_MAX_SIZE + 1] = "";
/**@}*/

//Translates a byte (8 bits) to 2 HEX Chars (4+4 bits).
#define TO_HEX(i) ( (i) <= 9 ? '0' + (i) : 'A' - 10 + (i) )

////////////////////////////////////////////////////////////////
// Variables used in IR Send/Receive
////////////////////////////////////////////////////////////////
/**
 * \defgroup ir_variables 1.4. Variables used in IR Send/Receive
 * @{
 */
/// IR RX Pin D6 GPIO12 in ESP8266 nodeMCU.
const uint8_t kRecvPin = 12;
/// Set the GPIO to be used to receive data.
IRrecv irrecv(kRecvPin);

/// IR TX D7 GPIO13 in ESP8266 nodeMCU.
const uint8_t kIrLed = 13;
/// Set the GPIO to be used to send data.
IRsend irsend(kIrLed);

/// Variable where a received message is stored to.
decode_results received_data;
/// Variable where a message is stored before being sent.
decode_results send_data;

/// Transmission and Reception Buffers Size.
#define MAX_PACKET_SEQUENCE 1024
/// Reception data buffer of 32bits/packet.
uint32_t rx_data_buffer[MAX_PACKET_SEQUENCE];
/// Transmission Data buffer of 32bits/packet.
uint32_t tx_data_buffer[MAX_PACKET_SEQUENCE];
/**@}*/
```

```
////////////////////////////////////////////////////////////////
// Variables used in IR Frame Control
////////////////////////////////////////////////////////////////
/** \defgroup frame_packet_types 1.5. Frame Packet Types
  * Variables used in IR Frame Control:
  * 3 bits for type of packet.\n
  * 8 bits for dest deviceID.\n
  * 32 bits for data.\n
  * 16 bits for CRC16 Checksum.\n
  * Type of Packet:\n
  * 0, 000 -> HELLO_ENCRYPTED.\n
  * 1, 001 -> HELLO.\n
  * 2, 010 -> HELLO_NEW.\n
  * 3, 011 -> DATA.\n
  * 4, 100 -> ACK.\n
  * 5, 101 -> DATA_END (Last Data Packet -> Checksum).\n
  * 6, 110 -> DATA_END_ENCRYPTED (Last Data Packet (encrypted) -> Checksum).\n
  * 7, 111 -> NACK - Not Used.\n
  * @{
  */
#define TYPE_HELLO_ENCRYPT 0b000
#define TYPE_HELLO 0b001
#define TYPE_HELLO_NEW 0b010
#define TYPE_DATA 0b011
#define TYPE_ACK 0b100
#define TYPE_DATA_END 0b101
#define TYPE_DATA_END_ENCRYPT 0b110
#define TYPE_NACK 0b111 // Not Used right now
/**@}*/

/**
 * \defgroup program_flags 1.6. FLAGS used by the program to distinguish between states
 * @{
 */
/// Hello Communication Flag. 1 means HELLO Exchange was done, so don't listen for more HELLO
Packets.
uint8_t FLAG_HELLO_DONE = 0;

/// Encryption Flag. 1 means Data is encrypted(DH Key Exchange). Default is 1.
uint8_t FLAG_ENCRYPT_DATA = 1;
/**@}*/

/**@}*/


// *******************************************************************

/**
 * \defgroup functions 2. Functions
 * Module with all the custom functions implemented in the controller source code
 * @{
 */

/**
 * \defgroup debug_functions 2.1. DEBUG Functions
 * Functions which are used for debugging purposes (printing logs and traces mainly)
 * @{
 */


/**
  *  DEBUG Function which prints in terminal the time elapsed in seconds.
  *   @param[in] long ref: Reference time where the counting period begins.
  */
void printTimeSeconds(long ref){
  double elapsed = (millis()-ref)/1000.0;
  Serial.printf(" - Time Elapsed: %.4f seconds.\n", elapsed);
```

```
}

/**
  * DEBUG function used to print in terminal information regarding received packets.
  *  @param[in] uint32_t header_capture: integer that represents the 32bits captured in a IR
frame.
  *  @param[in] uint16_t received_counter: integer that represents the current packet counter
in the data exchange.
  *  @param[in] uint8_t  received_type: integer that represents the type of packet that has
been received.
  */
void printCapturedSeq(uint32_t header_capture, uint16_t received_counter, uint8_t
received_type) {
  Serial.println("\nDEBUG: printCapturedSeq() HEX:");
  Serial.print("Captured: ");       Serial.println(header_capture, HEX);
  Serial.print("Counter: ");        Serial.println(received_counter, HEX);
  Serial.print("packetType: ");     Serial.println(received_type, HEX);
  Serial.println("*****\n");
}

/// DEBUG function used to print in terminal sent packet frame, represented in the global
variable send_data.
void printSentPacket() {
  Serial.println("printSentPacket() info:");
  //serialPrintUint64(send_data.value, BIN);
  //Serial.println("");
  serialPrintUint64(send_data.value, HEX);
  Serial.println("");
}

/// DEBUG function used to print in terminal received packet frame, represented in the global
variable received_data.
void printReceivedPacket() {
  Serial.println("printReceivedPacket() info:");
  //serialPrintUint64(received_data.value, BIN);
  //Serial.println("");
  serialPrintUint64(received_data.value, HEX);
  Serial.println("");
}

/**
  * DEBUG function used to print in terminal information regarding getRXFrameData() function
output.
  *  @param[in] uint8_t destID: integer that represents frame's Device Destination ID value.
  *  @param[in] uint8_t packet_type: integer that represents the current packet's type.
  *  @param[in] uint8_t is_from_controller: integer of which only the LS bit is used. 1 means
packet comes from controller device.
  *  @param[in] uint32_t payload: integer representing the payload data of the frame.
  */
void printRXPacketInfo(uint8_t destID, uint8_t packet_type, uint8_t is_from_controller,
uint32_t payload) {
  Serial.println("\nDEBUG: printRXPacketInfo()[BIN]: ");
  Serial.print("destID: ");            Serial.println(destID, BIN);
  Serial.print("packetType: ");        Serial.println(packet_type, BIN);
  Serial.print("isfromcontroller: ");  Serial.println(is_from_controller, BIN);
  Serial.print("payload: ");           Serial.println(payload, BIN);
  Serial.println("*****\n");
}

/// DEBUG function used to print stored EEPROM parameters: SSID, PSK, devID, number_nodes...
void printEEPROMData() {
  Serial.print("\nSSID: ");
  Serial.println((char *) ssid);
  Serial.print("PSK: ");
  Serial.println((char *) psk);
  Serial.print("DeviceID: ");
  Serial.println(devID);
```

```
  Serial.print("Control char. target/actual: ");
  Serial.print((char) CONTROL_EEPROM_TARGET_CHAR[0]);
  Serial.print("/");
  Serial.println((char) eeprom_control_char[0]);
  Serial.println("");
}
/**@}*/

/**
 * \defgroup aux_functions 2.2. Program Auxiliar Functions
 * Functions which are called to perform important processes during the execution of the main
functions
 * @{
 */

/**
 * Function which connects the device to the IoT Acess Point (a RaspPi IoT Hub).
 * It assigns a static Ip Address to the device, based on his Device ID: 192.168.100.devID.
 * AP Gateway is 192.168.100.1
 */
void connectStaticToAP() {
  IPAddress staticIP(192, 168, 100, devID); // ESP static IP address
  IPAddress gateway(192, 168, 100, 1);      // IP Address of Gateway (RaspPi)
  IPAddress subnet(255, 255, 255, 0);       // Subnet mask
  IPAddress dns(8, 8, 8, 8);                // DNS (Google)

  const char* deviceName = "IoT_Node_"+devID; // Descriptive device name
  WiFi.hostname(deviceName);


  char *psk_hex = (char *) malloc(PSK_MAX_SIZE+1);
  for(int i=0; i<PSK_MAX_SIZE; i++){
    psk_hex[i] = TO_HEX(psk[i]);
  } psk_hex[PSK_MAX_SIZE] = '\0';


  WiFi.disconnect();
  WiFi.persistent(false);
  WiFi.begin((char *) ssid, psk_hex);
  WiFi.config(staticIP, gateway, subnet, dns); // Make config effective

  Serial.println((char *) ssid);
  Serial.println((char *) psk_hex);

  WiFi.mode(WIFI_STA); // WiFi Station mode

  // Wait for connection
  Serial.print("Connecting to AP ");
  Serial.print((char *) ssid);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("\nConnected!");
}

/**
  *  Function which delays execution a variable random amount of time.
  *   @param[in] bool short_time: If true, the delay will be a short delay.
  */
void wait(bool short_time) {
  if(short_time)
    delay(150+(rand()%10)); //Wait 150 to 160 ms
  else{delay(2000+(rand()%1000));} // wait 2000 to 3000 ms
}
```

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

telecom BCN

```
/**
  * Function used to simplify the process of reading a single entry stored in the EEPROM.
  *  @param[in] uint8_t start_addr: integer that represents the initial address where the
information is stored.
  *  @param[in] uint8_t num_bytes: integer that represents the number of bytes (or addresses)
to read.
  *  @param[out] unsigned char* result: pointer to where the read output should be stored.
  */
void readEEPROM(uint8_t start_addr, uint8_t num_bytes, unsigned char* result) {
  EEPROM.begin(64);
  uint8_t index = 0;

  result[index] = (char) (EEPROM.read(start_addr+index));
  while (index<num_bytes-1) {
    index = index+1;
    result[index] = (char) (EEPROM.read(start_addr+index));
  }
  EEPROM.end();
}

/**
  * Function used to simplify the process of writing a single entry to the EEPROM.
  *  @param[in] uint8_t start_addr: integer that represents the initial address where the
information is stored.
  *  @param[in] unsigned char *data: pointer to where the read output should be stored.
  *  @param[in] uint8_t size: integer that represents the number of bytes (or addresses) to
read.
  */
void writeEEPROM(uint8_t start_addr, unsigned char *data, uint8_t size) {
  EEPROM.begin(64);
  uint8_t index = 0;
  EEPROM.write(start_addr+index, data[index]);
  while(index < size-1){ //data[index] != '\0' &&
    index = index+1;
    EEPROM.write(start_addr+index, data[index]);
  }
  EEPROM.commit();
  EEPROM.end();
}

/// Function used to initialize the EEPROM with the received information from controller:
<SSID,PSK> and NodeID.
void saveConfigEEPROM() {
  Serial.println("Saving Received Data From Controller into EEPROM...");

  writeEEPROM(ADDR_SSID, ssid, SSID_MAX_SIZE);
  writeEEPROM(ADDR_PSK, psk, PSK_MAX_SIZE);

  // Conversion from int to char array - devID of controller (1)
  unsigned char devID_char[1];
  devID_char[0] = devID;
  writeEEPROM(ADDR_DEV_ID, devID_char, 1);

  // Write control char 'Y' in 0xFF to tell device's EEPROM has data
  strncpy((char *) &eeprom_control_char, (char *) &CONTROL_EEPROM_TARGET_CHAR, 2);
  writeEEPROM(ADDR_CONTROL_CHAR, eeprom_control_char, 1);
}

/// Function used to read all EEPROM configuration values and load them in the program global
variables.
void loadConfigValuesFromEEPROM() {
  // Read SSID,PSK
  readEEPROM(ADDR_SSID, SSID_MAX_SIZE, ssid);
  readEEPROM(ADDR_PSK, PSK_MAX_SIZE, psk);

  // Read Node DeviceID
  unsigned char devID_char[2];
```

```
  readEEPROM(ADDR_DEV_ID, 2, devID_char);
  devID = devID_char[0];

  // Read Control Char
  unsigned char control_char[2];
  readEEPROM(ADDR_CONTROL_CHAR, 2, control_char);
  eeprom_control_char[0] = control_char[0];
}

/**
  * Function that performs the Data integrity Check for a burst of received packets.\n
  * It does a CRC32 check and determines if the data is valid.
  *  @param[in] uint16_t size: Number of received packets in the RX_buffer.
  *  @param[in] uint32_t rx_checksum: Checksum Received in the last data packet.
  * @return bool: True is data is valid. False if data is not valid.
  */
bool checkDataIntegrity(uint16_t size, uint32_t rx_checksum){
  uint32_t checksum = CRC32::calculate(rx_data_buffer, size);
  Serial.println("Checking Checksums...");
  Serial.printf("Received: %d | calculated: %d\n", rx_checksum, checksum);
  if(checksum == rx_checksum)
    return true;

  Serial.println("Checksums not matching!");
  return false;
}

/**
  * Function used to convert the info to send into a formatted frame ready to be sent.
  *   @param[in] uint8_t destID: device to which the packet is sent.
  *   @param[in] uint8_t packet_type: integer that represents the current packet's type.
  *   @param[in] uint32_t payload: integer representing the payload data of the frame.
  *  @return uint64_t: integer which represents the frame ready to be sent (up to 36 bits,
RC6 IR protocol).
  */
uint64_t buildTXFrame(uint8_t destID, uint8_t packet_type, uint32_t payload) {
  uint64_t data = 0ULL;
  if(packet_type==TYPE_ACK || packet_type==TYPE_NACK){
    data =
      static_cast<uint64_t>(payload & 0xFFF)     << 0  |
      static_cast<uint64_t>(0ULL)                << 12 | //Packet Sent From Node, always 0
      static_cast<uint64_t>(packet_type & 0b111) << 13;
  }
  else if(packet_type==TYPE_HELLO || packet_type==TYPE_HELLO_ENCRYPT ||
packet_type==TYPE_HELLO_NEW) {
    data =
      static_cast<uint64_t>(payload)     << 0  |
      static_cast<uint64_t>(0ULL)        << 24 | //Packet Sent From Node, always 0
      static_cast<uint64_t>(destID)      << 25 |
      static_cast<uint64_t>(packet_type) << 33;
  }
  else if(packet_type==TYPE_DATA || packet_type==TYPE_DATA_END_ENCRYPT ||
packet_type==TYPE_DATA_END) {
    data =
      static_cast<uint64_t>(payload)     << 0  |
      static_cast<uint64_t>(0ULL)        << 32 | //Packet Sent From Node, always 0
      static_cast<uint64_t>(packet_type) << 33;
  }

  //Serial.println("DEBUG: buildTXFRAME:");
    //serialPrintUint64(data, HEX);
    //Serial.println();
    //serialPrintUint64(data, BIN);
    //Serial.println("\n");
  return data;
}
```

```
/**
  * Function which processes the received frame stored in received_data global variable.\n
  * It processes the information of the frame and determines if the packet is valid.
  *   @param[in] uint16_t current_counter: represents the number of received packets in the
current burst.
  * @return uint32_t: integer which contains 2 different variables:\n
  *   Packet type (8bits)  in the 8 LSBs.\n
  *   New Counter (16bits) in the 16 MSBs. If NewCounter==current_counter_input+1, it means
the packet is valid. Everything else means packet was wrong and it get discarded.
  */
uint32_t getRXFrameData(uint16_t current_counter, uint32_t rand_hello_target=0) {
  uint64_t data = received_data.value;
  uint16_t nbits = received_data.bits;

  uint8_t destID = 0;
  uint8_t packet_type = 0;
  uint32_t payload = 0;
  uint8_t is_from_controller = 0;

  if(nbits==16) { // Short Frame -> TYPE ACK
    packet_type = (uint8_t) ((data>>13) & 0b111); // 3 bits
    payload = (uint32_t) (data & 0xFFF); // 12 bits
    is_from_controller = (uint8_t) ((data>>12) & 0b1 ); // 1 bit
  }
  else if(nbits==36){ // Long Frame -> TYPE HELLO, DATA, DATA_END
    packet_type = (uint8_t) ((data>>33) & 0b111); // Located in same place for all 36-bit
frames

    if(packet_type==TYPE_HELLO || packet_type==TYPE_HELLO_ENCRYPT) {
      payload = (uint32_t) (data & 0x00FFFFFF); // 24 bits
      is_from_controller = (uint8_t) ((data>>24) & 0b1 );
      destID = (uint8_t) (data>>25); //DestID only present in HELLO seqs
    }
    else if(packet_type==TYPE_HELLO_NEW) {
      payload = (uint32_t) (data & 0x00FFFFFF); // 24 bits
      is_from_controller = (uint8_t) ((data>>24) & 0b1 );
      destID = (uint8_t) (data>>25); //DestID only present in HELLO seqs
    }
    else if (packet_type==TYPE_DATA || packet_type==TYPE_DATA_END ||
packet_type==TYPE_DATA_END_ENCRYPT){
      payload = (uint32_t) (data & 0xFFFFFFFF); // 32 bits
      is_from_controller = (uint8_t) ((data>>32) & 0b1 );
    }
  }

  if(!is_from_controller){Serial.println("\n**a** (not coming from controller)");return -1;}

  uint16_t next_counter = current_counter + 1;
  uint32_t response_ok = packet_type | (((uint32_t) (next_counter))<<16);
  uint32_t response_ko = packet_type;

  //Serial.print("DEBUG: responseOK: ");
  //serialPrintUint64(response_ok, BIN);
  //Serial.println("\n\n");

  switch (packet_type) {
    case TYPE_ACK:
      //Serial.println("*Got ACK*");
        //Serial.printf("Payload: %d, current_counter: %d. Are equal: ", payload,
current_counter);
        //Serial.println((uint16_t) (payload)==current_counter);
      if ((uint16_t) (payload)==current_counter) { //ACKs should contain nº of packets
received
        Serial.printf("Got ACK number %d\n", current_counter);
        //printRXPacketInfo(destID, packet_type, is_from_controller, payload);
        return response_ok;
      }
```

```
    return response_ko;
    break;

  case TYPE_NACK:
    return response_ko;
    break;

  case TYPE_HELLO_ENCRYPT:
    if(FLAG_HELLO_DONE)
      return response_ko;
    //printRXPacketInfo(destID, packet_type, is_from_controller, payload);

    if(devID==0 && devID_aux>0) {
      Serial.println("Got a HelloEncrypt after NEW HELLO... ");
      if(payload==rand_hello_target) {
        FLAG_ENCRYPT_DATA = 1;
        return response_ok;
      }
      return response_ko;
    }

    if(devID>0) {
      Serial.println("Got a Hello Encrypt... ");
      if(payload!=devID) {
        Serial.printf("...but it is for another devID: %d\n", payload);
        return response_ko;
      }
      if(devID==destID) {
        devID_aux = devID;
        FLAG_ENCRYPT_DATA = 1;
        return response_ok;
      }
    }

    Serial.println("HELLO Encrypt not for me :(");
    return response_ko;
    break;

  case TYPE_HELLO:
    if(FLAG_HELLO_DONE)
      return response_ko;
    //printRXPacketInfo(destID, packet_type, is_from_controller, payload);

    if(devID==0 && devID_aux>0) {
      Serial.println("Got a Hello after NEW HELLO... ");
      if(payload==rand_hello_target) {
        FLAG_ENCRYPT_DATA = 0;
        return response_ok;
      }
      return response_ko;
    }

    if(devID>0) {
      Serial.println("Got a Hello... ");
      if(payload!=devID) {
        Serial.printf("...but it is for another devID: %d\n", payload);
        return response_ko;
      }
      if(devID==destID) {
        devID_aux = devID;
        FLAG_ENCRYPT_DATA = 0;
        return response_ok;
      }
    }

    Serial.println("it is not for me :(");
    return response_ko;
```

```
      break;

    case TYPE_HELLO_NEW:
      if(FLAG_HELLO_DONE)
          return response_ko;

      if(devID>0)
        Serial.printf("Broadcast New_Hello Detected, but I have my own id: %d\n", devID);
      if(destID>0)
        Serial.printf("Broadcast New_Hello Detected, but destID is not 0: %d\n", destID);
      if(payload==0)
        Serial.printf("Broadcast New_Hello Detected, but payload does not contain a newID:
%d\n", payload);

      if (devID>0 || destID>0 || payload==0)
        return response_ko;

      devID_aux = (uint8_t) payload;
      Serial.print("\nNew devID assigned from HELLO_NEW: ");
      Serial.println(devID_aux);

      return response_ok;
      break;

    case TYPE_DATA:
      if(current_counter>0 && rx_data_buffer[current_counter-1]==payload && payload>0)
        return response_ko;

      //Serial.printf("Got Data! counter: %d\n", current_counter);
      //printRXPacketInfo(destID, packet_type, is_from_controller, payload);
      rx_data_buffer[current_counter] = payload;
      return response_ok;
      break;

    case TYPE_DATA_END:
      if(current_counter==0)
        return response_ko;

      //Serial.printf("Got *LAST* Data! counter: %d\n", current_counter);
      if(checkDataIntegrity(current_counter, payload)) {
        FLAG_ENCRYPT_DATA = 0;
        return response_ok;
      }
      return response_ko;
      break;

    case TYPE_DATA_END_ENCRYPT:
      if(current_counter==0)
        return response_ko;

      //Serial.printf("Got *LAST* Data! counter: %d\n", current_counter);
      if(checkDataIntegrity(current_counter, payload)) {
        FLAG_ENCRYPT_DATA = 1;
        return response_ok;
      }
      return response_ko;
      break;

    default:
      return response_ko;
      break;
  }
}

/**
  * Function which encrypts the data before sending it over IR. It uses a 32-byte shared_key
to encrypt 32 bytes of information.\n
```

```
  *  As the key is unique in each session, there is no need to use a Init_Vector, as in this
case:\n
  *  len(data)==len(key), we can use a simple XOR bit comparison to encrypt/decrypt the Data.
  *   @param[in] byte *shared_key: Pointer to the session key used to encrypt/decrypt the
data.
  *   @param[in] byte *msg: Pointer to the message to be encrypted.
  *   @param[out] byte *encrypted: Pointer to where the encrypted result will be saved.
 */
void encryptData(byte *shared_key, byte *msg, byte *encrypted) {
  for(int i=0; i<SSID_MAX_SIZE; i++){
    encrypted[i] = msg[i]^shared_key[i];
    encrypted[i+SSID_MAX_SIZE] = msg[i+SSID_MAX_SIZE]^shared_key[i+SSID_MAX_SIZE];
  }
}

/**
  * Function which decrypts the data received over IR. It uses a 32-byte shared_key to
encrypt 32 bytes of information.\n
  * As the key is unique in each session, there is no need to use a Init_Vector, as in this
case:\n
  * len(data)==len(key), we can use a simple XOR bit comparison to encrypt/decrypt the Data.
  *   @param[in] byte *shared_key: Pointer to the session key used to encrypt/decrypt the
data.
  *   @param[in] byte *encrypted: Pointer to the message to be decrypted.
  *   @param[out] byte *decrypted: Pointer to where the decrypted result will be saved.
 */
void decryptData(byte *shared_key, byte *encrypted, byte *decrypted) {
  for(int i=0; i<SSID_MAX_SIZE; i++){
    decrypted[i] = encrypted[i]^shared_key[i];
    decrypted[i+SSID_MAX_SIZE] = encrypted[i+SSID_MAX_SIZE]^shared_key[i+SSID_MAX_SIZE];
  }
}

/**
  * Function that translates data char array ("ssid_1234") (Between 16 and 32 bytes) to
chunks of 32bits.
  *   @param[in] byte *char_data: Pointer to the data 8-bits array to be converted into a
24bits array.
  *   @param[in] int size: size of the data in bytes.
  *   @param[in] bool is_psk: Optimizes packet building in case the data is from the PSK, as
it uses only 4 bits for each character (50% efficiency otherwise). Default is false.
  * @return uint16_t: integer representing the number of data packets the TX_buffer contains.
  */
uint16_t buildTXDataBuffer(byte *char_data, int size) {
  uint16_t number_chunks = ceil((size*8)/32.0);
  uint32_t checksum;

  // Casting byte int array into uint32 array.
  // 1 uint32 will hold 4 bytes, as payload is 32bits.
  Serial.println("DEBUG: buildTXDataBuffer [HEX]:");

  for(int i=0; i<number_chunks; i++) {
      tx_data_buffer[i] =
        static_cast<uint32_t> (char_data[4*i])   << 0 |
        static_cast<uint32_t> (char_data[4*i+1]) << 8 |
        static_cast<uint32_t> (char_data[4*i+2]) << 16|
        static_cast<uint32_t> (char_data[4*i+3]) << 24;
      //serialPrintUint64(tx_data_buffer[i], HEX);
      //Serial.println();
  }

  // Avoid sending many 0's packet at the end of data.
  for(int i=0; i<number_chunks; i+=0){
    if(tx_data_buffer[i]==0) {
      for (int j=i; j<number_chunks-1; j++) {
        tx_data_buffer[j] = tx_data_buffer[j+1];
      }
```

```
      number_chunks -= 1;
    } else{i+=1;}
  }

  // Checksum calculation
  checksum = CRC32::calculate(tx_data_buffer, number_chunks);
  tx_data_buffer[number_chunks] = checksum;
  number_chunks += 1;

  //Serial.printf("Num of data packets: %d\n\n", number_chunks);
  return number_chunks;
}

/**
  * Core Function which listens for a connection from a Controller, receives the Wi-Fi
credentials and writes the data in the EEPROM.\n
  * It is called periodically, until correctly receiving the information from a Controller.\n
  * It includes several connection stages:\n
  *     HELLO_NEW: Handshake with a random proof for new devices, which allows concurrence
problems with multiple new devices answering the controller at the same time.\n
  *     HELLO: Normal handshake, performed for all devices. it also informs the node if the
transmission will be encrypted or not.\n
  *     KEY_EXCHANGE: If users selects Encryption in the Controller, there is a key exchange
stage where both nodes send each other their public keys.\n
  *     DATA_EXCHANGE: Receives the data (in clear or encrypted) until receiving final ACK
from node.
  * @return bool: True if data was received. False otherwise.
  */
bool waitForControllerCommunication() {
  Serial.println("DEBUG: waitForControllerCommunication2 begins - Waiting for HELLO
packet.");
  long ref_time = millis();
  uint32_t rnd_rep = 0;

  devID_aux = devID;

  FLAG_HELLO_DONE = 0;

  bool HELLO_PENDING = true;
  bool NEW_HELLO_PENDING = true;
  uint16_t packet_sequence = 0;
  long time0 = millis();
  irrecv.resume();

  if(devID==0) {
    Serial.println("*TIME: HELLO_NEW Starts");
    while (NEW_HELLO_PENDING) {
      ref_time = millis();
      delay(4);
      if(irrecv.decode(&received_data)) {
        if(received_data.decode_type==RC6) {
          uint32_t header_capture = getRXFrameData(packet_sequence);
          uint8_t received_type = (uint8_t) (header_capture);
          uint16_t received_counter = (uint16_t) (header_capture>>16);

          if( received_type==TYPE_HELLO_NEW && received_counter==1 ){
            NEW_HELLO_PENDING = false;
            printTimeSeconds(ref_time);
            Serial.println("HELLO_NEW from Controller Received!");

            rnd_rep = ESP8266TrueRandom.random(1, 1<<24) & 0x00FFFFFF;
            Serial.printf("Rnd Payload Gen.: %d\n", rnd_rep);

            send_data.value = buildTXFrame(CONTROLLER_ID, TYPE_HELLO_NEW, rnd_rep);
            irsend.sendRC6(send_data.value, 36);
          }
        }
```

```
        irrecv.resume();
      }
    }
    Serial.print("*TIME: HELLO_NEW Ends ");
    printTimeSeconds(ref_time);
  }

  time0 = millis();
  Serial.println("*TIME: HELLO Starts");

  while (HELLO_PENDING) {
    if(NEW_HELLO_PENDING) { ref_time = millis(); delay(30); }
    else { delay(123); }

    if(irrecv.decode(&received_data)) {
      if(NEW_HELLO_PENDING)
        irrecv.resume();
      if(received_data.decode_type==RC6) {
        uint32_t header_capture = getRXFrameData(packet_sequence, rnd_rep);
        uint8_t received_type = (uint8_t) (header_capture);
        uint16_t received_counter = (uint16_t) (header_capture>>16);

        if( (received_type==TYPE_HELLO || received_type==TYPE_HELLO_ENCRYPT) &&
received_counter==1 ) {
          HELLO_PENDING = false;
          printTimeSeconds(ref_time);
          Serial.println("HELLO from Controller Received!");
          Serial.printf("REceived_type %d\n", received_type);
          Serial.printf("Encrypt Flag: %d\n", FLAG_ENCRYPT_DATA);
          if(FLAG_ENCRYPT_DATA) { send_data.value = buildTXFrame(CONTROLLER_ID,
TYPE_HELLO_ENCRYPT, devID_aux); }
          else {                    send_data.value = buildTXFrame(CONTROLLER_ID, TYPE_HELLO,
devID_aux); }
        }
      }
    }

    if(!NEW_HELLO_PENDING) {
      delay(49);
      Serial.println("Sent NEW_Hello");
      irsend.sendRC6(send_data.value, 36);
      irrecv.resume();
    }

    if(!NEW_HELLO_PENDING && HELLO_PENDING && (millis()-time0 > 2500))
      return false;
  }

  Serial.print("*TIME: HELLO Received (while loop out) ");
  printTimeSeconds(ref_time);
  printTimeSeconds(time0);

  // HELLO MSG successfully received. Send Back a HELLO with our address as payload
  Serial.println("\nAnswering Controller with a HELLO Response");
  time0 = millis();

  for(int i=0; i<3; i++) {
    delay(55);
    irsend.sendRC6(send_data.value, 36);
  }
  //while(millis()-time0<270) {
  //  delay(55);
  //  irsend.sendRC6(send_data.value, 36);
  //}

  Serial.print("*TIME: HELLO Response Sent ");
  printTimeSeconds(ref_time);
```

```
  printTimeSeconds(time0);

  FLAG_HELLO_DONE = 1;



  // **************** HELLO DONE, Exchanging Curve PublicKeys ****************



  uint8_t mycurve_shared[32];
  uint8_t hiscurve_public[32];
  uint8_t mycurve_secret[32];

  if(FLAG_ENCRYPT_DATA) {
    Serial.println("Calculation Encrypting Keys...");

    // Generating all parameters for Elliptic Curve Exchange
    static const uint8_t basepoint[32] = {9};

    uint8_t mycurve_public[32];

    for(int i=0; i<32; i++) { mycurve_secret[i] = ESP8266TrueRandom.randomByte(); }
    curve25519_donna(mycurve_public, mycurve_secret, basepoint);
    //for(int i=0; i<32; i++) { Serial.printf("Mycurve_public Byte %d: %d\n", i,
mycurve_public[i]); }



    // *** Sending Device's PublicKey to Controller ***



    Serial.print("*TIME: Sending myPubKey ");
    time0 = millis();

    uint8_t packet_max_number = buildTXDataBuffer(mycurve_public, 32);
    bool ACK_PENDING = true;
    time0 = millis();
    uint8_t n_attempts = 0; // Return after 3 missing acks

    // Loop for Sending Device's PublicKey
    while(ACK_PENDING) {
      Serial.println("\nEntering Sending my PublicKey Loop...");
      for(int i=0; i<packet_max_number; i++){
        delay(5);
        if(i+1==packet_max_number){
          if(FLAG_ENCRYPT_DATA){ send_data.value = buildTXFrame(CONTROLLER_ID,
TYPE_DATA_END_ENCRYPT, tx_data_buffer[i]); }
          else {                 send_data.value = buildTXFrame(CONTROLLER_ID, TYPE_DATA_END,
tx_data_buffer[i]); }
        }
        else {                   send_data.value = buildTXFrame(CONTROLLER_ID, TYPE_DATA,
tx_data_buffer[i]); }

        Serial.printf("Packet number %d\n", i);
        irsend.sendRC6(send_data.value, 36);
      }

      Serial.print("*TIME: All PublicKey info sent");
      printTimeSeconds(ref_time);
      printTimeSeconds(time0);

      irrecv.resume();
      Serial.println("Waiting for ACK PublicKey");
      time0 = millis();
```

```
    while(ACK_PENDING) {
      delay(2);
      if(irrecv.decode(&received_data)) {
        if(received_data.decode_type == RC6) {
          uint32_t header_capture = getRXFrameData(packet_max_number);
          uint8_t received_type = (uint8_t) (header_capture);
          uint16_t received_counter = (uint16_t) (header_capture>>16);

          if( received_type==TYPE_ACK && received_counter==packet_max_number+1 ) {
            ACK_PENDING = false;
            Serial.println("\n\n**Burst ACK Received for PublicKey**\n");
          }
        }
        if(ACK_PENDING)
          irrecv.resume();
      }
      if(millis()-time0 > 300)
        break;
    }

    if(ACK_PENDING){
      n_attempts += 1;
      if(n_attempts>=4)
        return false;
    }
  }

  Serial.print("\n*TIME: myPubKey Sent | Waiting for Real Data ");
  printTimeSeconds(ref_time);
  printTimeSeconds(time0);
}



// **************** Curve Public Key Exchange Done, Data RX Starts Here ****************



bool IS_LAST_DATA = false;
time0 = millis();

while(!IS_LAST_DATA) {
  bool CHECKSUM_ERROR = false;
  packet_sequence = 0;
  irrecv.resume();

  if(millis()-time0>2000)
    return false;

  while(!IS_LAST_DATA && !CHECKSUM_ERROR) {
    delay(4);
    if(irrecv.decode(&received_data)) {
      if(received_data.decode_type == RC6) {
        uint32_t header_capture = getRXFrameData(packet_sequence);
        uint8_t received_type = (uint8_t) (header_capture);
        uint16_t received_counter = (uint16_t) (header_capture>>16);

        if( received_type==TYPE_HELLO && received_counter==packet_sequence+1 ){
          send_data.value = buildTXFrame(CONTROLLER_ID, TYPE_HELLO, devID_aux);
          delay(51);
          irsend.sendRC6(send_data.value, 36);
        }
        if( received_type==TYPE_DATA && received_counter==packet_sequence+1 ) {
          Serial.printf("Data %d received.\n", packet_sequence);
          packet_sequence++;
        }
```

```
          if(received_type==TYPE_DATA_END || received_type==TYPE_DATA_END_ENCRYPT) {
            if(received_counter==packet_sequence+1){
              Serial.printf("**LAST** Packet %d received.\n", packet_sequence);
              IS_LAST_DATA = true;
              packet_sequence++;
              if(received_type==TYPE_DATA_END_ENCRYPT) { Serial.println("*** Data End
Encrypt!!"); }

            } else { Serial.println("Checksum ERROR"); memset(&rx_data_buffer, 0,
packet_sequence); CHECKSUM_ERROR = true; }
          }
        }
        if(!IS_LAST_DATA)
          irrecv.resume();
      }
    }
  }

  Serial.print("*TIME: All DATA Received");
  printTimeSeconds(ref_time);
  printTimeSeconds(time0);

  Serial.println("Sending DATA ACK");
  send_data.value = buildTXFrame(CONTROLLER_ID, TYPE_ACK, packet_sequence);
  time0 = millis();

  for(int i=0; i<4; i++) {
    delay(5);
    irsend.sendRC6(send_data.value, 16); //ACK
  }
  //while(millis()-time0 < 180) {
  //  irsend.sendRC6(send_data.value, 16); //ACK
  //  delay(15);
  //}

  Serial.print("*TIME: ACK DATA Sent - Pre Saving Data");
  printTimeSeconds(ref_time);
  printTimeSeconds(time0);

  byte msg[32];
  for(int i=0; i<8; i++) { // 8 packets with info
    msg[4*i+0] = rx_data_buffer[i] >> 0*8;
    msg[4*i+1] = rx_data_buffer[i] >> 1*8;
    msg[4*i+2] = rx_data_buffer[i] >> 2*8;
    msg[4*i+3] = rx_data_buffer[i] >> 3*8;
    //Serial.printf("Buffer %d: ", i);
    //serialPrintUint64(rx_data_buffer[i], HEX);
    //Serial.println();
  }

  if(FLAG_ENCRYPT_DATA) {
    for(int i=8; i<packet_sequence-1; i++) { // PublicKey is 32bytes: packet_seq=8-15
      hiscurve_public[4*(i-8)+0] = (uint8_t) (rx_data_buffer[i] >> 0*8);
      hiscurve_public[4*(i-8)+1] = (uint8_t) (rx_data_buffer[i] >> 1*8);
      hiscurve_public[4*(i-8)+2] = (uint8_t) (rx_data_buffer[i] >> 2*8);
      hiscurve_public[4*(i-8)+3] = (uint8_t) (rx_data_buffer[i] >> 3*8);
      //Serial.printf("Buffer HisPublicKey %d: %d | %d | %d | %d\n", i, hiscurve_public[4*(i-
8)+0], hiscurve_public[4*(i-8)+1], hiscurve_public[4*(i-8)+2], hiscurve_public[4*(i-8)+3]);
      //Serial.println();
    }

    curve25519_donna(mycurve_shared, mycurve_secret, hiscurve_public);
    //for(int i=0; i<32; i++) { Serial.printf("Mycurve_shared Byte %d: %d\n", i,
mycurve_shared[i]); }

    Serial.print("*TIME: Key Saved and Secret Calculated.");
    printTimeSeconds(ref_time);
```

```
    printTimeSeconds(time0);

    Serial.println("Message was Encrypted! Decrypting...");
    decryptData(mycurve_shared, msg, msg);
  }


  for(int i=0; i<SSID_MAX_SIZE; i++){
    ssid[i] = (unsigned char) (msg[i]);
    psk[2*i+0] = (unsigned char) ( (msg[i+SSID_MAX_SIZE] >> 0*4) & 0x0F);
    psk[2*i+1] = (unsigned char) ( (msg[i+SSID_MAX_SIZE] >> 1*4) & 0x0F);
  }

  Serial.print("*TIME: DATA Saved");
  printTimeSeconds(ref_time);
  printTimeSeconds(time0);

  Serial.println("*** SSID Received ***");
  Serial.println((char *) ssid);
  Serial.println("*** PSK Received ***");
  for(int i=0; i<32; i++){ Serial.printf("%c", TO_HEX(psk[i])); }
  Serial.println('\n');

  printTimeSeconds(ref_time);

  Serial.println("All Data Received, saving all Information in EEPROM.");

  devID = devID_aux;
  saveConfigEEPROM();

  Serial.println("Info in Variables After Reading EEPROM:");
  loadConfigValuesFromEEPROM();
  printEEPROMData();

  Serial.println("Exiting Function waitForControllerCommunication()");
  return true;
}
/**@}*/

// *********************************************************************

/**
 * \defgroup main_functions 2.3. Program Main Functions
 * Functions which must be implemented by an ESP8266. They execute the main logic of the
program with the aid of the previous functions
 * @{
 */

/// Setup function which checks EEPROM and, if not initialized, starts listening for a
controller until receiving the information, writing it into EEPROM.
void setup() {
  Serial.begin(115200);

  while (!Serial)       // Wait for the serial connection to be establised.
    delay(50);
  Serial.println();

  srand(time(NULL)); // Initialize Random Gen Seed

  //Uncomment this to reset memory by removing 'Y' control char
  //unsigned char a[2] = {'A', '\0'};
  //unsigned char id[2] = {0, '\0'};
  //writeEEPROM(ADDR_CONTROL_CHAR, a, 1);
  //writeEEPROM(ADDR_DEV_ID, id, 1);

  readEEPROM(ADDR_CONTROL_CHAR, 2, eeprom_control_char);
  Serial.println("Memory before loading EEPROM:");
```

```
  printEEPROMData();

  loadConfigValuesFromEEPROM();
  Serial.println("Memory after loading EEPROM:");
  printEEPROMData();

  // *******************************************************************

  irrecv.enableIRIn();  // Preparamos el receptor IR
  irsend.begin();       // Preparamos el emisor IR
  irrecv.resume();

  if(eeprom_control_char[0]!=CONTROL_EEPROM_TARGET_CHAR[0] || devID==0)
    Serial.println("\n\n Memory not Setup or devID not assigned, waiting for controller
connection");
  else
    Serial.println("Device has an ID assigned, trying to connect to AP");


  //If control char does not exist or it is 'Y' but device has no ID, start listening to
controller
  while ( (eeprom_control_char[0]!=CONTROL_EEPROM_TARGET_CHAR[0])
      ||(eeprom_control_char[0]==CONTROL_EEPROM_TARGET_CHAR[0] && devID==0) ) {
    waitForControllerCommunication();
    wait(1);
  }
  Serial.println("\nInfo Received from controller!\n");


  Serial.println("Setting Up connection with Access Point...");
  while(WiFi.status() != WL_CONNECTED) {
    Serial.println("Connecting to AP...");
    connectStaticToAP();
    delay(100);
  }

  Serial.printf("Connection with AP successful! Try pinging device: ping 192.168.100.%d\n",
devID);
}


// *********************************************************************


/// Loop() function not used. Just sleep.
void loop() {
  delay(50);
}
/**@}*/

/**@}*/
/**@}*/
```

Annex III - RaspPi WIFI_SETUP.PY Code

```
import os
import sys
import subprocess
import re

'''
Script which modifies all the necessary files in the system in order to get a customized Access Point
running.
It can be called manually, but it will usually be called by another credential extraction script. The
Access Point
configuration is persistent and should start working after a device reboot.
To call it from a terminal, type: python3 wifi_setup.py <ssid> <key>"
Author: Rafael de la Rosa Ortiz
'''


FAIL = b'Contrase\xc3\xb1a: \r\nsu: Fallo de autenticaci\xc3\xb3n\r\n'

WLAN0_LINES = [
    'interface wlan0\n',
    '   static ip_address=192.168.100.1/24\n',
    '   nohook wpa_supplicant\n',
]
DNSMASK_LINES = [
    'interface=wlan0\n',
    'dhcp-range=192.168.100.200,192.168.100.250,255.255.255.0,24h\n',
]
HOSTAPD_LINES = [
    'interface=wlan0\n',
    'driver=nl80211\n',
    'ssid=IoT_test\n',
    'hw_mode=g\n',
    'channel=7\n',
    'wmm_enabled=0\n',
    'macaddr_acl=0\n',
    'auth_algs=1\n',
    'ignore_broadcast_ssid=0\n',
    'wpa=2\n',
    'wpa_passphrase=ThisIsMyWPAPassword\n',
    'wpa_key_mgmt=WPA-PSK\n',
    'wpa_pairwise=TKIP\n',
    'rsn_pairwise=CCMP\n'
]
LOCALRC_LINE = "sudo iptables-restore < /etc/iptables.ipv4.nat\n"

# Main function of the script: it will install all needed packets, and configure all necessary files in
order to create
#   an Access Point with the credentials passed as first and second argument.
# It can be called as many times as needed, it will destroy previous AP and create a new one.
# It is persistent, Access Point will still be running after a device reboot.
# It needs sudo access, if not provided, it will call itself with sudo privileges to be able to install
packages and
#   modify some files
def main():
    if os.geteuid() == 0:
        print("Root Access OK")
    else:
        print("No Root... Executing again with sudo")
        subprocess.call(['sudo', 'python3', *sys.argv])
        sys.exit(1)
    if len(sys.argv) < 3:
        print("Wrong format!")
        print("     Format: python3 wifi_setup.py <ssid> <key>")
        exit(1)

    myssid = sys.argv[1]
    mypsk = sys.argv[2]

    global HOSTAPD_LINES
    HOSTAPD_LINES[2] = 'ssid='+myssid+'\n'
    HOSTAPD_LINES[10] = 'wpa_passphrase='+mypsk+'\n'
```

```
    print("\nConfiguring Custom Access Point with the following parameters:")
    print("     SSID: "+myssid)
    print("     PSK:  "+mypsk)
    print()

    # Creates file to save the current ssid and password for user consulting
    with open("/home/accesspoint_params.txt", 'w') as f:
        f.write('Current Access Point Credentials:\n')
        f.write("ssid: "+myssid+'\n')
        f.write("psk: "+mypsk+'\n')

    os.system('sudo apt install -y dnsmasq hostapd')
    os.system('sudo systemctl stop dnsmasq')
    os.system('sudo systemctl stop hostapd')

    os.system('sudo apt install -y iptables-persistent')


    cmd = 'sudo grep -n "interface wlan0" /etc/dhcpcd.conf'
    resp = '0'
    try:
        resp = subprocess.check_output(cmd, shell=True).decode('utf-8')
    except subprocess.CalledProcessError:
        pass

    lineNum = -1
    overwrite = False

    if re.search('interface wlan0', resp):
        print(resp)
        print("Interface exists, overwriting it...")

        lineNum = int(re.split(':', resp, 1)[0])
        overwrite = True

    if overwrite:
        lines = None
        with open("/etc/dhcpcd.conf", 'r') as f:
            lines = f.readlines()
            if len(lines) > lineNum:
                for l in range(0, len(WLAN0_LINES)):
                    try:
                        lines[lineNum+l-1] = WLAN0_LINES[l]
                    except IndexError:
                        lines.append(WLAN0_LINES[l])

        with open("/etc/dhcpcd.conf", 'w') as f:
            f.writelines(lines)

    else:
        with open("/etc/dhcpcd.conf", 'a+') as f:
            f.writelines(['\n']+WLAN0_LINES)

    os.system('sudo service dhcpcd restart')

    with open("/etc/dnsmasq.conf", 'w') as f:
        f.writelines(DNSMASK_LINES)

    os.system('sudo systemctl reload dnsmasq')

    with open("/etc/hostapd/hostapd.conf", 'w') as f:
        f.writelines(HOSTAPD_LINES) # ToDo set custom SSID and PSK!

    os.system('''sudo sed -i 's+#DAEMON_CONF=""+DAEMON_CONF="/etc/hostapd/hostapd.conf"+g' 
/etc/default/hostapd''')

    os.system('sudo systemctl unmask hostapd')
    os.system('sudo systemctl enable hostapd')
    os.system('sudo systemctl start hostapd')
    os.system('sudo systemctl start dnsmasq')

    os.system("sudo sed -i 's/#net.ipv4.ip_forward=1/net.ipv4.ip_forward=1/g' /etc/sysctl.conf")

    os.system("sudo iptables -F INPUT")
    os.system("sudo iptables -F OUTPUT")
```

```
    os.system("sudo iptables -F FORWARD")
    os.system("sudo iptables -t nat -F")

    os.system("sudo iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE")

    os.system('sudo sh -c "iptables-save > /etc/iptables.ipv4.nat"') # for rc.local, not working
currently as rc.local gets executed before iptables service
    os.system('sudo sh -c "iptables-save > /etc/iptables/rules.v4"') # for iptables-persistent package


    with open("/etc/rc.local", 'r') as f:
        lines = f.readlines()

    exists = False
    for line in lines:
        if re.search(LOCALRC_LINE, line):
            exists = True

    if not exists:
        lines[lines.index("exit 0\n")] = LOCALRC_LINE
        lines.append("exit 0\n")

    with open("/etc/rc.local", 'w') as f:
        f.writelines(lines)

    print('\n Configuration completed. The WiFi AP should be up and running. It will persist if device
gets rebooted.')

if __name__ == "__main__":
    main()
```

Annex IV - RaspPi CAMERA_SCAN.PY Code

```
import cv2
import re
import numpy
import zbar
import os

'''
Script which scans QR Codes and looks for a valid format to extract the credentials (ssid and psk) from
the decoded info
and calls the wifi_setup.py script to setup a new Access Point in the Raspberry Pi that is executing the
code.
To call it from a terminal, type: python3 camera_scan.py"

Author: Rafael de la Rosa Ortiz
'''



# Class which converts the RaspberryPi Camera in a QR Code Scanner. It will look for a QR encoding a
<SSID,PSK> pair and
# call the wifi_setup.py script in order to setup an Access Point with those credentials.
class Scanner:
    # Initializes the camera for QR scanning
    def __init__(self):
        cv2.namedWindow("w1", flags=cv2.WINDOW_NORMAL)

        self.capture = cv2.VideoCapture(-1)
        self.capture.set(cv2.CAP_PROP_FRAME_WIDTH, 960)     # 960px width
        self.capture.set(cv2.CAP_PROP_FRAME_HEIGHT, 720)    # 720px height

        # Scanner only detects codes with black pixels over white background.
        # We invert image 50% of the times to capture all possibilities
        self.invert = False

        # Call scan function from class Scanner.
        # It executes and infinite loop until a valid QR is received
        self.scan()

    def scan(self):
        received = False

        while not received:
            # Take a photo to analyze it
            ret, self.frame = self.capture.read()

            aframe = numpy.asarray(self.frame[:, :])

            # Switch between invert ON and invert OFF
            if self.invert:
                self.frame = cv2.bitwise_not(self.frame)
            self.invert = not self.invert

            # Convert pixel to gray scale
            imgray = cv2.cvtColor(self.frame, cv2.COLOR_BGR2GRAY)

            raw = str(imgray.data)

            # Initialization ZBAR class that looks for valid QR
            scanner = zbar.ImageScanner()
            scanner.parse_config('enable')

            width = int(self.capture.get(cv2.CAP_PROP_FRAME_WIDTH))
            height = int(self.capture.get(cv2.CAP_PROP_FRAME_HEIGHT))

            # Create a ZBAR image object and search for a QR
            imageZbar = zbar.Image(width, height, 'Y800', raw)
            scanner.scan(imageZbar)

            ssid = None
            psk = None

            # In case one or several QR codes are detected, look for a valid SSID and PSK
```

```python
            # SSID is valid if it starts with the prefix 'IOT_'
            for symbol in imageZbar:
                print 'decoded', symbol.type, 'symbol', '"%s"' % symbol.data
                if "IOT_" in symbol.data:
                    cv2.destroyWindow("w1") # Stops the scanner, as valid QR was found

                    ssid, psk = re.split(':\n', symbol.data, 1)
                    print("Found valid Credentials. Calling the wifi_setup script:")
                    print('python3 /home/pi/Documents/wifi_setup.py %s %s' % (ssid, psk))
                    os.system('python3 /home/pi/Documents/wifi_setup.py %s %s' % (ssid, psk))
                    exit(1) # Exit program

            # Show image in the preview window (only if called locally, not from remote console)
            cv2.imshow("w1", aframe)

            # Wait 100ms between captures
            cv2.waitKey(100)


# Class calling: Initializes it and turns camera on for QR scanning
scan = Scanner()
```

Annex V - RaspPi SERIAL_READER.PY Code

```
import serial
import re
import time
import os

'''
Script which extracts the credentials (ssid and psk) from a ESP8266 IoT Controller by reading the serial
USB port,
and calls the wifi_setup.py script to setup a new Access Point in the Raspberry Pi that is executing the
code.
To call it from a terminal, type: python3 serial_reader.py"

Author: Rafael de la Rosa Ortiz
'''

myport = '/dev/ttyUSB0'  # Linux Serial
# myport = 'COM6'  # Windows Serial

ser = None

# Serial initialization. If device is not connected, display a message and wait for the user to plug it
in
while ser is None:
    try:
        ser = serial.Serial(port=myport, baudrate=115200)
    except serial.serialutil.SerialException:
        print('Connect Device to Serial port (/dev/ttyUSB0)')
        time.sleep(0.25)

ssid = None  # line format to extract info: b'SSID: ABCDEF \r\n'
psk = None

# Keep reading Serial until a valid psk is extracted (psk is always extracted after ssid)
while psk is None:
    line = ser.readline()  # Reads bytes from the serial port
    print(line)

    if re.match(b'SSID: ', line) and len(line) > 10:  # A shorter line means ssid is empty
        ssid = re.split(b'SSID: ', line)[1]
        ssid = re.split(b'\r\n', ssid)[0].strip().decode("utf-8")

    if re.match(b'PSK: ', line) and ssid is not None:
        psk = re.split(b'PSK: ', line)[1]
        psk = re.split(b'\r\n', psk)[0].strip().decode("utf-8")

    time.sleep(0.1)

print('python3 /home/pi/Documents/wifi_setup.py %s %s' % (ssid, psk))
os.system('python3 /home/pi/Documents/wifi_setup.py %s %s' % (ssid, psk))
```

Glossary

- **AP**: Access Point. Device which connects wireless devices, establishing a wireless network.
- **BER**: Bit Error Ratio. It measures the number of bit errors divided by the total number of transferred bits in a given interval.
- **CPU**: Central Processing Unit. Brain of a computer where most calculations take place.
- **DC**: Direct Current. Electrical current which flows in one single direction.
- **Debian**: It is a Linux free and open-source software distribution.
- **DHCP**: Dynamic Host Configuration Protocol. It is a network management protocol in charge of dynamically assigning IP addresses to devices in a network, so they can communicate with other devices in different networks.
- **Doxygen**: Tool for writing software reference documentation.
- **EC**: Error Correction. In a QR Code of a given size, the Error Correction determines the degree of redundancy of the encoded information. A higher EC would make the QR more robust, but it also reduces the amount of total data that could be coded.
- **EEPROM**: Electrically Erasable Programmable Read-Only Memory. It is a type of non-volatile memory used in computers to store small amounts of data.
- **ESP8266**: It is a low-cost Wi-Fi microchip with full TCP/IP stack and microcontroller capability produced by manufacturer Espressif Systems.
- **GPIO**: General Purpose Input/Output. It is an uncommitted digital signal pin on an electronic circuit board whose behavior is controllable by the user at run time.
- **HTML**: HyperText Markup Language. It is the standard markup language for documents designed to be displayed in a web browser
- **ICMP**: Internet Control Message Protocol. It is a supporting protocol in the Internet protocol suite. It is used by network devices to send error messages and operational information indicating success or failure when communicating with another IP address.
- **IDE**: Integrated Development Environment. It is a software application that provides comprehensive facilities to computer programmers for software development.
- **IEEE**: Institute of Electrical and Electronics Engineers. It is a professional association for electronic engineering and electrical engineering.
- **IoT**: Internet of Things. It is a system of interrelated computing devices that are provided with unique identifiers and the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction.

- **IP**: Internet Protocol. It is the principal communications protocol in the Internet protocol suite for relaying datagrams across network boundaries.
- **IR:** InfraRed. It is electromagnetic radiation (EMR) with longer wavelengths than those of visible light.
- **ISP**: Internet Service Provider. It is an organization that provides services for accessing, using, or participating in the Internet.
- **KRACK**: Key Reinstallation Attack. It is a severe replay attack (a type of exploitable flaw) on the Wi-Fi Protected Access protocol that secures Wi-Fi connections.
- **LED**: Light-Emitting Diode. It is a semiconductor light source that emits light when current flows through it.
- **NAT**: Network Address Translation. It is a method of remapping one IP address space into another by modifying network address information in the IP header of packets while they are in transit across a traffic routing device.
- **Nonce**: It is an arbitrary number that can be used just once in a cryptographic communication.
- **OLED**: Organic Light-Emitting Diode. It is a LED in which the emissive electroluminescent layer is a film of organic compound that emits light in response to an electric current.
- **OS**: Operative System. It is system software that manages computer hardware, software resources, and provides common services for computer programs.
- **PIN**: Personal Identification Number (do not confuse with hardware input/output -GPIO-pin). It is a numeric or alpha-numeric password used in the process of authenticating a user accessing a system.
- **PSK**: Pre-Shared Key. It is a shared secret which was previously shared between the two parties using some secure channel before it needs to be used.
- **QR Code**: Quick Response Code. It is a two-dimensional barcode which encodes information that can be scanned using a specific QR scanner or a phone camera.
- **RAM**: Random Access Memory. It is a form of computer memory that can be read and changed in any order, typically used to store working data and machine code.
- **RaspPi**: Raspberry Pi. It is a series of small single-board computers to promote teaching of basic computer science in schools and in developing countries.
- **RC6**: IR protocol developed by Philips as a semi-proprietary consumer IR remote control communication protocol for consumer electronics. It is the successor of the RC5.

- **RISC**: Reduced Instruction Set Computer. It is a computer whose instruction set architecture allows it to have fewer cycles per instruction (CPI) than a complex instruction set computer (CISC).

- **RNG**: Random Number Generator. It is a device that generates a sequence of numbers or symbols that cannot be reasonably predicted better than by a random chance.

- **RX**: Receiver.

- **SSH**: Secure Shell. It is a cryptographic network protocol for remotely operating network services securely over an unsecured network.

- **SSID**: Service Set Identifier. It is a sequence of up to 32 octets included in the header of a packet in a wireless network to identify it as part that network.

- **TOR Network**: The Onion Router Network. It is free and open-source software for enabling anonymous communication.

- **TX**: Transmitter.

- **USB**: Universal Serial Bus. It is an industry standard that establishes specifications for cables and connectors and protocols for connection, communication and power supply between computers, peripheral devices and other computers.

- **WPA**: Wi-Fi Protected Access. It is a security protocol and a security certification program developed by the Wi-Fi Alliance to secure wireless computer networks. There are currently three versions: WPA, WPA2 and WPA3.

- **WPS**: Wi-Fi Protected Setup. It is a network security standard to create a secure wireless home network.

- **XOR**: eXclusive OR. It is a logical operation that outputs true only when inputs differ.