BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

SECURE PROVISIONING OF IOT DEVICES

BEZPEČNÉ ZPROVOZNĚNÍ IOT ZAŘÍZENÍ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. PETR RUSIŇÁK**
AUTOR PRÁCE

SUPERVISOR — **Mgr. KAMIL MALINKA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2021

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2020/2021

Master's Thesis Specification

Student: **Rusiňák Petr, Bc.**
Programme: Information Technology Field of study: Information Technology Security
Title: **Secure Provisioning of IoT Devices**
Category: Security
Assignment:

1. Study existing secure IoT provisioning solutions and protocols, supply a comparison of various approaches.
2. Design own secure IoT provisioning system using ESP32 platform, based on the ESP-NOW protocol (Espressif's proprietary connectionless WiFi protocol):
   a) study the ESP-NOW protocol details,
   b) suggest suitable way of authorising devices for subsequent WiFi network access (certificates, whitelisting). For this very case it is legitimate to "sacrifice" one device for providing specific provisioning service (ie the provisioning is not strictly general, which is also given by use ESP-NOW),
   c) mind possible future extensions (eg application firmware bootstrapping), make whole system modular,
   d) supply a proof-of-concept code before starting real implementation.
3. Implement the solution as designed in item 2.
4. Demonstrate a function of the system, evaluate your results and suggest possible improvements/extensions.

Recommended literature:

- ESP NOW (https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/network/esp_now.html)
- ESP32 devboards tutorial (build/flash/monitor) - https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/index.html
- IoT provisioning principles - https://www.electronicdesign.com/technologies/iot/article/21126707/runtime-provisioning-of-security-credentials-for-iot-devices
- Existing implementations of ESP32-based provisioning - https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/provisioning/index.html

Requirements for the semestral defence:

- Items 1 and 2.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Malinka Kamil, Mgr., Ph.D.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2020
Submission deadline: May 19, 2021
Approval date: November 11, 2020

Abstract
With the increasing number of IoT devices sold each year, many large-scale applications using IoT devices are emerging. This creates a demand for secure and easy-to-use device provisioning protocols, as it is not feasible to spend a tremendous amount of time configuring the devices. The aim of this thesis is to create a secure provisioning protocol that will configure the network credentials on an out-of-the-box IoT device automatically. The implemented protocol uses a dedicated Configurator device that stores the network credentials of all compatible devices from a given administrative domain. The Configurator device will provide the network credentials to unconfigured devices upon request, assuming they are able to authenticate with the Configurator. The protocol uses public-key cryptography to verify the identity of the devices during. This protocol was implemented using ESP32 devices, and the connectionless ESP-NOW protocol is used to communicate with the unconfigured devices.

Abstrakt
S ohledem na stále rostoucí počty prodaných IoT zařízeních se postupně začínají objevovat projekty, ve kterých jsou IoT zařízení použity ve stovkách až tisících. Tyto projekty však z časových důvodů neumožňují ruční konfiguraci každého zařízení zvlášť, čímž vzniká poptávka po protokolech, které dokáží rychle, ale přitom i bezpečně, nastavit nové IoT zařízení. Cílem této práce je vytvořit protokol, který umožní automatický přenos přihlašovacích údajů k Wi-Fi síti do nově zakoupeného IoT zařízení. Navržený protokol používá speciální konfigurační zařízení, ve kterém budou uloženy přihlašovací údaje všech zařízeních kompatibilních s tímto protokolem v rámci dané administrativní domény, a které bude poskytovat tyto přihlašovací údaje nenakonfigurovaným IoT zařízením za předpokladu, že je možné ověřit jejich identitu. K ověření identity nenakonfigurovaných zařízení je použita asymetrické kryptografie. Protokol byl implementován pomocí IoT zařízeních ESP32, přičemž ke komunikaci mezi nenakonfigurovanými je využit nespojovaný komunikační protokol ESP-NOW.

Keywords
IoT, provisioning, IoT device provisioning

Klíčová slova
IoT, provisioning, zprovoznění IoT zařízení

Reference
RUSIŇÁK, Petr. *Secure Provisioning of IoT Devices.* Brno, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Mgr. Kamil Malinka, Ph.D.

Rozšířený abstrakt

V dnešní době již IoT zařízení nejsou jen záležitostí nadšenců, ale nacházejí své využití i v rámci masových průmyslových nasazení. Instalace stovek nebo tisíců nových IoT zařízení však s sebou nese nová úskalí, na která nejsou současná IoT zařízení připravena.

V současnosti instalace nového IoT zařízení často probíhá tak, že nové IoT zařízení při svém prvním spuštění vytvoří vlastní dočasnou Wi-Fi síť, ke které se musí uživatel připojit z jiného zařízení (typicky počítače nebo mobilu). Po připojení k této síti dojde k přesměrování na speciální stránku s formulářem, který žádá o vyplnění přístupových údajů k Wi-Fi síti, ke které se bude IoT zařízení v budoucnu připojovat. Po odeslání tohoto formuláře si IoT zařízení uvedené údaje uloží, ukončí vysílání vlastní Wi-Fi sítě a připojí se k zadané síti. V kapitole 3 však jsou uvedeny i jiné existující přístupy k prvotní konfiguraci zařízení, např. pomocí mobilní aplikace nebo agregačního zařízení, které se snaží sjednotit co nejvíce protokolů používaných různými IoT zařízeními do jednoho rozhraní.

Dále je v kapitole 4 rozebrán nadcházející standard Wi-Fi Easy Connect, jehož cílem je vytvořit jednotný postup pro snadné přidávání nových zařízení (ať už klientů nebo přístupových bodů) do existujících Wi-Fi sítí. Pro přidání nového zařízení do sítě jsou v rámci tohoto standardu zapotřebí dvě zařízení: konfigurované zařízení a konfigurátor, kde konfigurátor je zřízení, které na vyžádání poskytne konfigurovanému zařízení přístupové údaje do Wi-Fi sítě. Tento standard je však v současnosti ve fázi příprav, a proto není podporován současnými Wi-Fi zařízeními. Uvedený standard je navíc příliš komplexní pro jeho přímé využití v rámci této práce, neboť pokrývá velké množství případů užití, ale některé jím definované koncepty zde budou převzaty.

Obdobně jako je tomu v případě nadcházejícího standardu Wi-Fi Easy Connect, tato práce definuje konfigurační protokol pro nastavení nového zařízení. Tento protokol využívá speciální konfigurační zařízení, ve kterém jsou uloženy přihlašovací údaje všech IoT zařízení, které chtějí využívat tento protokol. Konfigurace nového IoT zařízení pak probíhá tak, že toto nové zařízení při svém prvním spuštění automaticky vyhledá konfigurační zařízení a požádá jej o poskytnutí přístupových údajů k Wi-Fi síti, které mu toto konfigurační zařízení po předchozí autentizaci poskytne.

Vlastní konfigurační protokol se tak skládá z několika fází. Pro účely popisu tohoto protokolu, nechť *stanice* je nenakonfigurované IoT zařízení, které využívá služeb konfigurátoru za účelem získání přístupových údajů k Wi-Fi síti a *konfigurátor* je zařízení, které na vyžádání poskytuje stanicím jejich přístupové údaje k Wi-Fi síti.

Ještě před spuštěním tohoto konfiguračního protokolu je však nutné, aby obě zařízení splňovala určité požadavky. Zaprvé, všechna zařízení (stanice i konfigurátor) musí mít vygenerovaný pár soukromého a veřejného RSA klíče, který bude použit k ověření identity daného zařízení. Dále je nutné, aby konfigurátor obsahoval seznam přihlašovacích údajů k Wi-Fi síti všech stanic, přičemž každá stanice je identifikována SHA-256 otiskem jejího veřejného klíče. A nakonec, je-li vyžadována oboustranná autentizace obou zařízení, je zapotřebí stanici předem poskytnout veřejný klíč konfigurátoru.

Konfigurační protokol je pak zahájen ze strany stanice, která pravidelně vysílá všesměrové zprávy informující konfigurátor o její existenci. Tato zpráva mj. obsahuje SHA-256 otisk veřejného klíče stanice, což umožňuje konfigurátor dopředu určit, zda má smysl reagovat na danou zprávu (pokud konfigurátor daný otisk nezná, předpokládá, že je zprává určena pro jiný konfigurátor v okolí a ignoruje). Tyto oznamovací zprávy stanice vysílá až do okamžiku, kdy od konfigurátoru obdrží požadavek na zaslání svého veřejného klíče. Na tuto zprávu stanice odpoví zasláním svého kompletního veřejného klíče. Po přijetí odpovědi

si konfigurátor ověří, že získaný klíč odpovídá jeho otisku, který znal předem. Pokud ano, protokol postoupí do autentizační fáze.

Průběh autentizační fáze je závislý na tom, zda stanice vyžaduje oboustrannou autentizaci, neboť identita stanice je ze strany konfigurátoru ověřována vždy, zatímco ověření konfigurátoru ze strany stanice je v rámci tohoto protokolu volitelné. Vlastní průběh autentizace je pak založen na principu výzva-odpověď, kdy jedno zařízení vygeneruje výzvu zašifrovanou veřejným klíčem druhého zařízení, která je protistranou dešifrována a zaslána zpět po zašifrování veřejným klíčem prvního zařízení. V případě neaktivní oboustranné autentizace je tento postup zjednodušen na přímé odeslání přihlašovacích údajů k Wi-Fi síti po zašifrování veřejným klíčem stanice.

Po dokončení autentizační fáze konfigurátor odešle stanici přístupové údaje k Wi-Fi síti. Vzhledem k běžné délce přihlašovacích údajů k Wi-Fi síti protokol neustanovuje mezi zařízeními sdílené tajemství pro symetrickou šifru, ale přihlašovací údaje jsou asymetricky zašifrovány veřejným klíčem stanice. Stanice si následně tyto údaje uloží v perzistentní paměti, aby nebylo nutné při dalším spuštění tohoto IoT zařízení nutné tyto údaje opět získávat pomocí tohoto protokolu.

Pro implementaci stanice i konfigurátoru byly použity IoT zařízení ESP32-S2-Saola-1. V rámci implementace protokolu bylo také nutné vyřešit fakt, že stanice s konfigurátorem nemohou navzájem komunikovat prostřednictvím cílové Wi-Fi sítě, neboť stanice ještě nezná její přístupové údaje. Toto bylo vyřešenou použitím nespojovaného protokolu ESP-NOW, který umožňuje komunikaci mezi dvěma ESP32 zařízeními jen na základě MAC adres. Interně je tento protokol implementován výrobcem pomocí IEEE 802.11 Management rámců.

Secure Provisioning of IoT Devices

Declaration

I hereby declare that this Masters's thesis was prepared as an original work by the author under the supervision of Mr. Malinka. The supplementary information was provided by Mr. Vychodil from Espressif Systems. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Petr Rusiňák
May 17, 2021

Acknowledgements

I would like to thank both Mr. Malinka and Mr. Vychodil for the consultations they provided me during the course of the whole academic year, and the insights they gave me into the inner working of IoT devices.

Contents

1 Introduction **3**

2 Problem definition **4**

3 Existing solutions **6**
3.1 Methodology . . . 6
3.2 Network discovery protocols . . . 6
3.3 Gateway-oriented solutions . . . 7
3.4 Device Cloud Middleware . . . 8
3.5 Cloud-oriented solutions . . . 8
3.6 Implementation in ESP-32 . . . 9
3.7 Chapter summary . . . 9

4 Wi-Fi Easy Connect standard **12**
4.1 Enrollee and Configurator roles . . . 12
4.2 Initiator and Responder roles . . . 13
4.3 Mutual authentication . . . 14
4.4 DPP prerequisites . . . 14
4.5 Provisioning Protocol . . . 14
4.5.1 Bootstrapping phase . . . 14
4.5.2 Authentication phase . . . 15
4.5.3 Configuration phase . . . 16
4.5.4 Access phase . . . 17

5 ESP-32 **18**
5.1 Basic features . . . 18
5.2 Development toolkit . . . 18
5.3 Build system . . . 19
5.4 Menuconfig . . . 20
5.5 ESP-NOW . . . 21
5.6 Cryptographic functions . . . 22
5.7 eFuse . . . 22
5.8 Flash encryption and secure boot . . . 23
5.9 Non-volatile storage . . . 24
5.10 FreeRTOS features . . . 25
5.10.1 Timers . . . 25
5.10.2 Tasks . . . 25
5.10.3 Queues . . . 26

6 Analysis 28

7 Design 29
7.1 Authentication of devices 30
7.2 Enrollee's prerequisites 31
7.3 Configurator's prerequisites 32
7.4 Message fragmentation 32
7.5 Messages 33
7.5.1 Presence announcement 34
7.5.2 Public key request 34
7.5.3 Public key reply 35
7.5.4 Authentication request 35
7.5.5 Authentication reply 36
7.5.6 Connection details message 36

8 Implementation 38
8.1 Key generation and storage 38
8.2 Network credentials storage 39
8.3 Cryptography wrapper 40
8.4 Network stack 41
8.4.1 Wi-Fi initialization 41
8.4.2 Task for processing ESP-NOW messages 41
8.4.3 Sending a message 41
8.4.4 Receiving a message 42
8.5 STA (Enrollee) operation 43
8.5.1 The initial (null) state 43
8.5.2 Broadcasting presence 44
8.5.3 Wait for authentication request 44
8.5.4 Wait for connection details 44
8.6 Configurator operation 44
8.6.1 Processing the Presence announcement message 45
8.6.2 Processing the Public key reply message 45
8.6.3 Processing the Authentication reply message 45
8.7 Program configuration and execution 46

9 Evaluation 47
9.1 Fulfillment of requirements 47
9.2 User experience 48
9.3 Security considerations 49
9.3.1 Attacks on the protocol when mutual authentication is enabled 49
9.3.2 Attacks on the protocol when mutual authentication is disabled 49
9.3.3 Attacks on application deployment 49
9.3.4 Attacks on the Enrollee device 50
9.3.5 Attacks on the Configurator device 50
9.4 Possible extensions 50

10 Conclusion 52

Bibliography 54

Chapter 1

Introduction

Over the past two years, the number of Internet of Things (IoT) devices sold annually has increased from 3.9 million in 2018 to 4.8 million in 2019 and is expected to reach 5.8 million in 2020 [21]. IoT devices are now not used by enthusiasts and home users only but they also play a vital role in many large-scale business applications, as evidenced by many research papers on the topic such as [25, 32].

However, utilizing IoT devices in large-scale environments brings new challenges when deploying hundreds or thousands of new devices to an IoT network. New devices must be provided with network credentials so they can connect to the target network, and their application software often needs to be configured based on the use-case of each device. In home environments, this usually means flashing each device with application-specific firmware and configuring it manually. However, this approach deems unacceptable for large-scale applications because a considerable amount of time must be spent configuring each individual device. Some manufacturers provide tools that simplify this provisioning process but as will be shown in Chapter 3, these tools are far from perfect.

The aim of this thesis is to design and implement a new zero-touch provisioning protocol that will automatically provide a new IoT device with network credentials in a secure manner when the new device is first booted up. The protocol uses an Enrollee-Configurator scheme, where the new IoT device (Enrollee) communicates with a dedicated Configurator device to obtain its network configuration details (the SSID name and network credentials). In this implementation, ESP32-S2 microchips are used as both Enrollee and Configurator devices. To overcome the issue that the Enrollee device has no Wi-Fi connectivity yet, a connectionless protocol ESP-NOW is used to communicate between the devices.

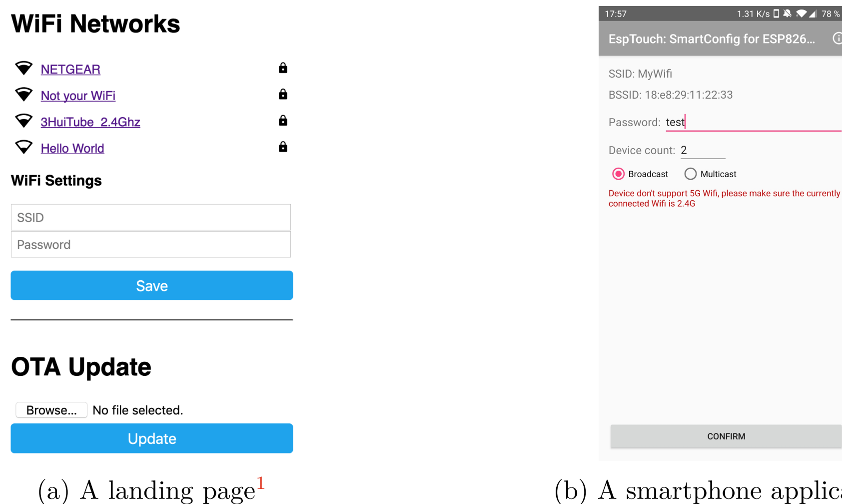
The proposed provisioning protocol consists of three phases: first, the Enrollee locates the Configurator using broadcast messages. Second, the Configurator verifies the Enrollee's identity using public-key cryptography (and optionally vice-versa). And third, the Configurator sends the Enrollee the network configuration details.

The thesis is structured as follows. First, Chapter 2 further defines the problem at hand. Then, Chapter 3 overviews existing theoretical and practical solutions to the problem, and Chapter 4 overviews an upcoming standard Wi-Fi Easy Connect, that can be leveraged when implementing a new provisioning protocol. This is followed by Chapter 5, which introduces the chosen ESP32 chip and its features. Chapter 6 analyses the state-of-the-art explained in the previous chapters and describes the implications for our approach. After that, Chapter 7 describes the design of the proposed provisioning protocol, and Chapter 8 highlights its implementation. In Chapter 9, the protocol will be evaluated from the ease-to-use and the security points of view. Finally, Chapter 10 concludes the thesis.

Chapter 2

Problem definition

As will be explained in detail in Chapter 3, the current solutions to the provisioning of IoT devices are somewhat cumbersome. In some cases, it is necessary to connect the IoT device to a computer using a USB cable and flash its firmware manually before the device can be used. Other unconfigured IoT devices will create a Wi-Fi Access Point, and the user is expected to connect to this network from their computer or smartphone. Upon connecting, the user is redirected to a landing page (such as the one shown in Figure 2.1a), where they enter the network credentials that will be used by the IoT device to connect to an existing Wi-Fi network.



(a) A landing page[1]

(b) A smartphone application

Figure 2.1: Examples of interfaces for setting up Wi-Fi credentials to IoT devices

The IoT device chosen for the implementation of this thesis is the **ESP32-S2-Saola-1** developed by Espressif Systems. The reason for choosing this device is that the ESP32 devices are quite affordable, and the S2-Saola-1 variant offers all necessary features for this task, especially a debugger interface, Wi-Fi module, and an accelerator for cryptographic operations. It should be noted that the author of this theses has received two of these devices from the aforementioned company free of charge.

The ESP32 devices currently offer a quite specific method for the provisioning of IoT devices with network credentials. Instead of forcing the user to configure the connection

[1]Source of image: https://esphome.io/components/captive_portal.html

details on each device individually, the user downloads a smartphone app[2] shown in Figure 2.1b. In this app, the user enters the Wi-Fi credentials (the app uses the same SSID the smartphone is currently connected to, thus only the pre-shared key must be entered), and upon clicking a button, the app will broadcast these details to all nearby unconfigured ESP devices. More details on this app will be given in Section 3.6. However, the main fallback of this solution is that the app may broadcast the Wi-Fi credentials to an unauthorized device. The app also does not allow sending a different set of credentials to each device.

Thus, there is a demand for a new approach. Chapter 4 will introduce an upcoming standard Wi-Fi Easy Connect from the Wi-Fi Alliance that provides a new approach to this issue, however, the standard is too complex because it solves a broader issue.

The existing approached will be further analysed in Chapter 6. However, there are some requirements for the implementation of the provisioning protocol that can be defined now:

- There should be a dedicated configurator device that will store all necessary network credentials, and provide them to all legitimate IoT devices, rather than forcing the user to configure the credential details on each device separately.
- As most IoT devices have little to no user interface, the implementation should not require any unnecessary user action on the IoT device that is provisioned.
- The provisioning protocol should be secure. It must not be possible for an unauthorized IoT device to obtain the network credentials. Other security vulnerabilities should be mitigated as well.
- The provisioning protocol must anticipate that the newly provisioned device has no internet connectivity, as the device did not receive the network credentials yet. Thus, an alternative way for the newly provisioned device and the configurator device to communicate must be found.
- The IoT device should only engage in the provisioning protocol if it had not received the network credentials yet.

[2]The app is available at https://apps.apple.com/us/app/espressif-esptouch/id1071176700 for iOs or https://play.google.com/store/apps/details?id=com.khoazero123.iot_esptouch_demo for Android

Chapter 3

Existing solutions

This chapter provides an overview of current approaches to the provisioning of IoT devices. First, Section 3.1 describes the methodology used to identify relevant literature sources. Then, Section 3.2 shows how some network discovery protocols can be leveraged to provide devices with network configuration. Sections 3.3 and 3.4 show state-of-the-art approaches to the provisioning of IoT devices, based on the gateway-oriented approach and the device cloud middleware respectively. Next, Section 3.5 gives examples of how the provisioning is done when installing a new virtual device in a cloud solution. Section 3.6 gives more details on how the current provisioning process is implemented in ESP32. And finally, Section 3.7 summarizes all aforementioned approaches and states what problems need to be considered when creating a new provisioning platform.

3.1 Methodology

In order to find all relevant approaches to IoT provisioning, this chapter will be based on both peer-reviewed academic sources and web articles. When reviewing academic sources, the ACM, IEEE Explore, and SpringerLink databases will be considered first. Every search term will be always looked up in all three databases, and the most relevant results among all databases will be used. If the number or quality of results is not sufficient, either other literature databases will be searched using tools like Google Scholar, or the search term will be modified.

Web articles will be used to gather information about commercial IoT solutions. This will typically include websites of IoT device manufacturers and documentation of their products, or third-party tutorials on IoT devices.

3.2 Network discovery protocols

In 2010, when IoT was just emerging, Patrick et al. suggested in their book *Vision and Challenges for Realising the Internet of Things* [31] to use existing network discovery protocols to obtain network connection details if the IoT device does not have these details hardcoded. They suggested using Bonjour, WS-Discovery, UPnP, or any similar protocol.

Bonjour is a zero-configuration protocol developed by Apple that allows automatic addressing and naming of devices, and service discovery. This protocol was designed to allow new devices, such as printers, to easily connect to a new network with zero interaction from the user apart from plugging the Ethernet cable in and powering up the device.

After the device obtains its IP configuration, the protocol will announce this new device's presence to all computers in the local area network (LAN). Thus, the protocol omits the need to configure the printer with IP configuration, and all workstations are not required to manually add this new device in their operating system's settings or to install the printer's drivers. [4]

The main disadvantage of this protocol is the way it handles addressing. If the new device's IP address is not issued using DHCP, the protocol expects that the device will use a link-local address only, which may not provide full internet access. The protocol also does not offer any convenient way to provide network credentials to the new device in case of a Wi-Fi connection. [4]

For device naming and service discovery, the protocol sends DNS-format messages over IP multicast. These messages are referred to as Multicast DNS (or mDNS) and they can be used to resolve local hostnames to IP addresses as well as to find servers that provide a particular service [4]. This protocol is supported by all major operating systems.

The **UPnP** protocol also expects devices to obtain their IP configuration through DHCP or by using link-local addresses [6]. The **WS-discovery** protocol supports service discovery only and assumes the devices are already provided with network configuration [29].

3.3 Gateway-oriented solutions

Many papers [5, 7, 23, 27, 28], especially those that focus on interconnecting IoT devices into a cloud network, propose to use a gateway device to interact with a group of IoT devices. The gateway device is responsible for aggregating multiple IoT devices in a single LAN segment by periodically reading data from all IoT devices in its segment, post-processing them, and sending the summarized data to a cloud computer. Thus, the configuration of connection details to the cloud server is set on the gateway device only, rather than on all IoT devices. In case the connection to the cloud server is disrupted, IoT devices will still be able to communicate with each other as long as they are connected to the same gateway, and the service provided by this IoT network might be accessible in a limited way. The gateway device may offer additional features based on the specific implementation, such as a management tool that allows the configuration of IoT devices that are connected to it.

One implementation of such gateway can be found in [23]. This gateway was designed for home users and supports a wide range of protocols and device types, including UPnP and ZigBee. The gateway acts as a proxy that allows unified access to all IoT devices in the household regardless of the different protocols that are used by individual IoT devices. To achieve this, the gateway has a specific implementation for each protocol and features a modular system that allows adding support for new protocols if needed in the future. However, when discovering new devices, the gateway uses the same type of discovery the original IoT device has. If the IoT device requires manual configuration on its first startup, the gateway will ask for the same configuration details, although the gateway provides its own user interface for this purpose, which might be easier to use. [23]

A different implementation is shown in [7]. This implementation recognizes the issues with managing many gateway devices in large-scale applications, such as managing smart-city networks. The work focuses on creating a single API that encapsulates the APIs provided by the gateways. The API leverages the concept of *software-defined IoT units*, which uses late-bound runtime policies. Instead of flashing each end-device with firmware specific to its runtime application, all devices are provided with a *bootstrap container* that is the same for all devices with compatible hardware architecture. When the device boots, the

bootstrap container will bound the device with software components based on its runtime application, and re-bound any software components on runtime if application requirements change. If the IoT device does not have the current version of any software component, the framework contains an API that allows the device to download the missing dependencies from the gateway. However, this presumes that the gateways implement this API. [7]

3.4 Device Cloud Middleware

Another approach to overcome the gateway devices' heterogeneity is to create a Device Cloud Middleware, as presented in [24]. This approach uses a three-layer architecture, that consists of the *Physical Space*, *Runtime Space*, and *Social Space* layers. All physical devices are present in the *Physical Space* layer, including both IoT devices and aggregator devices. The *Runtime Space* layer is responsible for data exchange between all interested parties. Finally, the *Social Space* layer represents the end-users of the system. [24]

The Device Cloud Middleware is located between the *Physical Space* and the *Runtime Space* and is responsible for transforming data from one format to another. The paper proposes to use existing devices that support automatic IoT device integration. This refers to devices that are used as gateway devices in the gateway-oriented approach, however, rather than creating clusters of devices that share the same gateway, there should be no fixed associations between devices in this approach. Instead, any gateway device can be used for any communication, as long as the gateway is able to interact with all interested parties. [24]

3.5 Cloud-oriented solutions

The problem of provisioning of new devices is also solved when creating new virtual devices in the cloud, regardless of whether a virtual IoT device is created, or a full-stack machine is installed. This section will first describe a solution used by a major Linux distribution, and then a solution used by Amazon Web Services will be shown.

Ubuntu offers a `cloud-init` package that is pre-installed on live server images. A cloud provider is able to send arbitrary user data to the operating system during machine startup. The `cloud-init` package retries this data, parses them, and runs actions present in the user configuration string. It is also possible to send a bash script through this user data, which the package will execute, meaning that is possible to make any change of the system through this configuration, albeit not in a zero-touch manner. [30]

Amazon Web Services (AWS) offer the Amazon Certificate Manager service that can be used to provision a new IoT device. To provision a new device, the device must first generate an asymmetric public and private key pair. Then, the device will generate a certificate signing request (CSR), that consists of non-sensitive data only, such as device and organization name, its location, and the signature of this CSR. This CSR is sent to Amazon Certificate Manager, which must decide its eligibility using an out-of-band channel, for example by comparing the device's unique identifier (UUID) with a whitelist. If allowed, the Manager will sign the certificate and send it to the device. The device will then be able to use this certificate to connect to the network. [26]

3.6 Implementation in ESP-32

Following the rather theoretical approaches to IoT provisioning given in the preceding sections, this section demonstrates concrete solutions used by the ESP32 microchip that is used for the implementation of the new zero-touch provisioning protocol that is the subject of this thesis. This device supports two provisioning protocols, Protocol Communication (Protocomm) and SmartConfig.

The **protocomm** protocol provides an extensible API a developer can use to create their own provisioning protocol by utilizing pre-defined features or by implementing use-case-specific features themselves. The protocol consists of IoT-device-side code and smartphone application, with both device-side code and phone application being open source in case their default behavior needs to be modified. The two can communicate using Wi-Fi or Bluetooth Low Energy (only the Wi-Fi approach will be considered from now on). [9]

In the Wi-Fi approach, the IoT device creates a temporary Wi-Fi Access Point (AP) that the smartphone can connect to. This will establish a secure session between the IoT device and the smartphone. Then, the smartphone app is used to provide the device with network connection details, such as SSID, username, and password. These details are transmitted using the protocomm protocol and saved into the device. This completes the provisioning process. On subsequent boots, the device will use the saved network credentials to connect to the Wi-Fi network. [9]

The **SmartConfig** protocol is a provisioning protocol developed by Texas Instruments. Just like protocomm, the protocol uses a smartphone app to configure the device. Instead of establishing a connection between the smartphone and the IoT device, the smartphone app broadcasts the network connection details. The IoT device is able to receive these details and use them to configure itself. Thus, this solution is easier to use because it is not necessary to connect to the IoT device manually, but it is less secure because the communication is prone to sniffing by an attacker. [8]

3.7 Chapter summary

An overview of approaches shown in the previous sections is shown in Table 3.1. The first column *Solution* contains the name of the solution or another description that identifies it. The column *IP configuration* identifies the main concept that the solution uses to provide an unconfigured device with enough information that it can connect to the target network and an obtain IP address. The *Other features* column lists other significant features the solution provides to further the provisioning process, such as provisioning the device with runtime application or having a management tool that simplifies the configuration of devices from different manufacturers.

As can be seen in the *IP configuration* column, there are provisioning solutions ([29, 24]) that do not expect that the device might not know enough information to connect to a network, but this may easily be the case if the device is supposed to connect to a secured Wi-Fi network. This is because these solutions focus on other tasks, such as application deployment ([24]) or service discovery ([29]). Other frameworks ([4, 6]) do anticipate this issue, but they rely on standard network mechanisms like DHCP or link-local addressing to solve it. This assumption is correct for wired connections, but it makes the process hard to work with in the case of wireless networks.

Among solutions that can work well in wireless environments, some similarities in network details provisioning can be observed. Most solutions ([23, 30, 9, 8]) use an external

Solution	IP configuration	Other features
Bonjour [4]	link-local address	service discovery over multicast
UPnP [6]	link-local address	device discovery
WS-discovery [29]	none	service discovery
Home gateway [23]	additional GUI	device management
Gateway with software-defined IOT units [7]	via bootstrapping	device management
Device Cloud Middleware [24]	none	device aggregation
Ubuntu [30]	configuration file	any configuration
AWS [26]	certificates	none
ESP protocomm [9]	smartphone app	can be implemented manually
ESP SmartConfig [8]	smartphone app	none

Table 3.1: Overview of existing IoT provisioning solutions

device that acts as a configurator. This configurator sends the network connection details to the IoT device using a pre-determined, typically out-of-band, protocol. The configurator can be implemented in many ways, including a desktop application ([23]), a smartphone application ([9, 8]), and a configuration file that is sent by the cloud service provider to the device ([30]). The bootstrapping solution ([7]) requires that the target the IoT device will be connected to (either Wi-Fi AP or Gateway device) supports a specific bootstrapping protocol that will be used to provide configuration details. Finally, the AWS solution ([26]) uses public-key cryptography to generate a certificate signing request to an external certificate manager. If the request is fulfilled, the IoT device will end up with a client certificate that can be used to connect to the network.

Thus, an ideal scheme for provisioning of network details should follow the following aspects:

- **Security** – the protocol needs to be secure. The network connection details must not be transmitted in a way that would allow an attacker to learn network credentials. An imposter device also must not be able to start the provisioning process and receive working connection details from the device responsible for issuing these details. Also, the case when a legitimate IoT device's provisioning request would be handled by an attacker-provided device, either to gain control over the IoT device or to perform a man-in-the-middle attack, must be considered.

- **Zero configuration** – ideally, the protocol should not require any interaction from the user. When user interaction is required, interoperability issues will arise, because every device manufacturer implements its own user-interfaces tools and protocols, often with different setup tools for different product lines of devices, and thus mass-configuration of many devices becomes cumbersome. If devices require zero interaction from the user, the issue that each device uses a different provisioning protocol becomes irrelevant. However, this is not possible for security reasons, because the scheme needs to known which IoT devices are allowed to join the network. But it is still encouraged to decrease the user interaction as much as possible. For example, an IoT device that has no graphical interface can avoid user interaction altogether, and the authorization-related configuration can be done on an external configurator device, such computer or smartphone.

- **Interoperability** – as some degree of user configuration must be preserved, the protocol should adhere to existing standards, when requesting user interaction, to ensure a single tool can be used to configure all devices. After initialization, the user application should also provide data in a standardized way, although this is out of the scope of this thesis.

When considering *other features*, many solutions ([23, 7, 30, 9]) either natively support or can be extended to support the deployment of application software. This can be done by creating a bootstrapping API that can bound artifacts at runtime ([7]), or by creating a tool that runs device-specific commands to update the application software (e.g. by flashing) remotely ([23, 30, 9]). However, as many standalone tools for application deployment exist, it may not be necessary to integrate application deployment into the provisioning protocol. Instead, the device could perform an over-the-air (OTA) update on its first boot after the provisioning of network credentials is completed, assuming the device's manufacturer provides an API for OTA updates.

Chapter 4

Wi-Fi Easy Connect standard

Alongside the existing approaches to provisioning described in the previous chapter, there is currently a new *Wi-Fi Easy Connect* standard in development. As the name of the standard suggests, it aims to make it easier to connect new devices to a Wi-Fi network. As of now, the standard is in a draft stage and subject to change, thus no commercially available solutions are implemented so far. However, many concepts introduced by this standard can be used when designing a new provisioning protocol, and thus this chapter aims to summarize key concepts of the upcoming standard.

First, Section 4.1 explains the overall concept of the approach used in this standard, and the *Enrollee* and the *Configurator* roles of individual devices in the architecture. Next, Section 4.2 explains the roles of *Initiator* and *Responder*, and how they differ from the *Enrollee* with *Configurator*. After that, Section 4.3 describes the difference between imprinting and mutual authentication, and Section 4.4 shows the preconditions that must be satisfied before the device provisioning protocol (or **DPP**, in short) can be started. And finally, the device provisioning protocol and its phases will be described in Section 4.5.

4.1 Enrollee and Configurator roles

As shown in Figure 4.1, the standard defines two roles of devices: *Enrollee* and *Configurator*. Note that both the workstation (or **STA**) and the access point (**AP**) it wants to connect to are considered to be *Enrollees*.

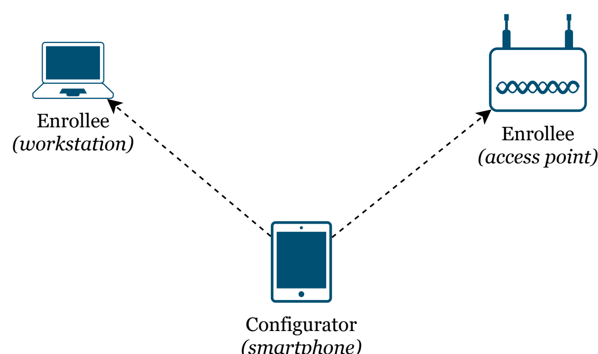


Figure 4.1: Device roles in Wi-Fi Easy Connect architecture

In the initial state, no links between any devices are formed, i.e. the workstation does not know how to connect to the access point, and the *Configurator* has no information about any of the *Enrollees*. During the device provisioning process (DPP), the *Configurator* will interact with all *Enrollees* as represented by the dashed lines in Figure 4.1. The goal of the DPP is to provision all devices with network credentials. In the scenario shown in Figure 4.1, the STA should be able to connect to the AP once DPP is completed.

From the user's perspective, the DPP in this scenario consists of using the smartphone (*Configurator*) to scan two QR codes – one that is displayed on the screen of the STA after clicking a *„join a new network"* button in the STA's network settings, and one that is printed on a label on the AP. Both QR codes can be scanned in any order. Once both codes are scanned, the STA connects to the AP automatically, and the configuration is completed.

A ***Configurator*** device is responsible for the registration of new devices into the network. A network compatible with this standard needs to have at least one configurator device, but the standard allows for multiple configurators in one network. It is advised that a device with a graphical interface is used as the *Configurator*, as the *Configurator* may provide features such as setting the access level of individual Enrollees, or revoking an Enrollee's access to the network if needed. Due to the frequent need to scan QR codes of the *Enrollees* (although not mandatory), using a smartphone as the *Configurator* may be a feasible choice. [33]

An ***Enrollee***, on the other hand, is a device that wants to take part in the network. It can either be an endpoint device, such as a computer, that wants to connect to a new network or a network device such as an access point. The provisioning protocol is not designed to connect endpoint devices only, but it can be used to install a new access point in the network as well. The protocol does not differentiate between endpoint devices and access points for the most part. When the two need to be handled specifically, endpoint devices are referred to as *STA Enrollee* and access points are referred to as *AP* in the standard. [33]

4.2 Initiator and Responder roles

While Figure 4.1 in the previous section shows that the DPP is started by the *Configurator*, this is not required by the standard. Depending on the use case, either *Enrollee* or *Configurator* may start the DPP. To identify what device started the DPP, the protocol defines the **Initiator** and the **Responder** roles.

When a provisioning process is started, one of the communicating devices plays the role of the *Initiator* and the other one plays the role of the *Responder*, i.e. either the *Configurator* plays the role of *Initiator* and the *Enrollee* plays the role of *Responder* or vice versa. The roles are determined by the device that stars the Authentication phase of the provisioning protocol, as this device always becomes the Initiator. Note that the Authentication phase is the second phase of the provisioning protocol, not the first, as will be shown in Section 4.5. [33]

4.3 Mutual authentication

The DPP supports mutual authentication, which may be either enabled or disabled based on the security requirements of the given scenario. If mutual authentication is enabled, both the *Initiator's* and the *Responder's* identity must be verified for the DPP to proceed. [33]

If the mutual authentication is disabled, only the *Responder's* identity is verified by the *Initiator*, while the *Responder* takes an imprinting approach towards the *Initiator*. This means the *Responder* implicitly trusts the first *Initiator* it encounters. Once the DPP is completed, the *Responder* should accept further provisioning requests only from the first *Initiator* it encountered, with a possibility to reset this association by the user. [33]

For example, consider a scenario where a configurator is setting up a Wi-Fi connection on a computer using the DPP. The *Configurator* takes the role of *Initiator*, and the computer *(Enrollee)* takes the role of *Responder*. If mutual authentication is disabled, only the computer's identity will be verified by the configurator. As will be shown in Section 4.4, this requires that the computer's public key is transmitted to the configurator using an out-of-band channel, for example by scanning a QR code that is displayed on the computer by the configurator. If mutual authentication is enabled, the computer will also need to verify the configurator's identity. Once again, this requires that the configurator's public key is transmitted to the computer using an out-of-band channel, for example by user input.

4.4 DPP prerequisites

Before the DPP can be initialized, the following preconditions must be satisfied. First, all devices (both *Enrollees* and *Configurators*) must contain a unique pair of the public and the private key that can be used for asymmetric cryptography generated by the elliptic curve cryptography (ECC) algorithm.

Second, the Initiator must obtain the Responder's public key, for example by scanning it from a QR code. For security reasons, the key must be obtained using an out-of-band channel. If mutual authentication is enabled, the Responder needs to know the Initiator's public key beforehand too. [33]

4.5 Provisioning Protocol

The **Device Provisioning Protocol** (DPP) is a protocol introduced by the Wi-Fi Easy Connect standard, that allows connecting new devices to a network with minimal interaction from the user with regards to the security of all concerned devices. DPP consists of four phases, Bootstrapping phase, Authentication phase, Configuration phase, and Access phase. These phases are executed sequentially and are explained in the subsections below. [33]

An example of an exchange of messages between the Initiator and the Responder during the DPP is shown in Figure 4.2. This example assumes no messages are lost in transit and all authentication attempts are successful. Because of the complexity of the messages, the diagram does not show data fields of individual messages. Notable fields will be described in the subsections below.

4.5.1 Bootstrapping phase

During the bootstrapping phase, one of the devices (Enrollee or Configurator) may periodically broadcast DPP presence announcement messages. By sending these messages,

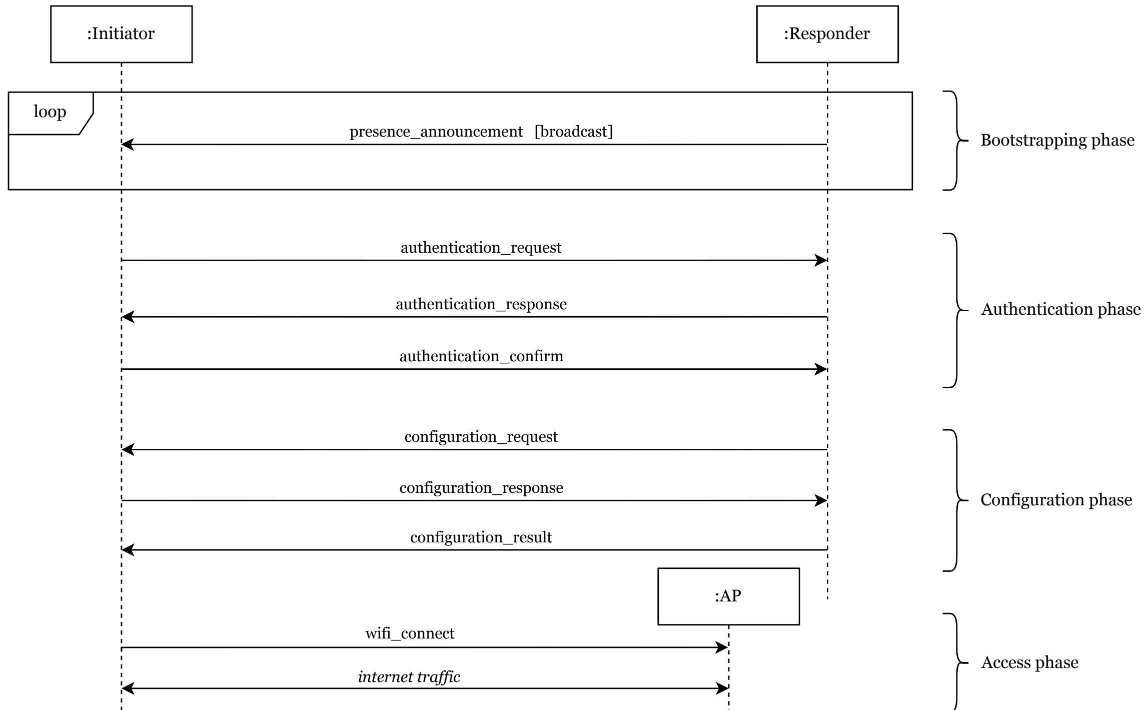


Figure 4.2: Example of the flow of messages during the device provisioning process

the device announces that is ready to engage in the DPP, and that it expects the other device to initialize the Authentication phase (and thus the announcing device will play the role of Responder). If the device wants to play the Initiator role, it may proceed to the Authentication phase without sending DPP presence announcements. The Responder may choose not to send the presence announcement messages and only wait for an authentication request message from the Initiator, and thus the Bootstrapping phase may be skipped altogether. [33]

4.5.2 Authentication phase

The authentication phase (shown in Figure 4.3) is responsible for the establishment of trust between the Initiator and the Responder. To achieve this, the Initiator's and the Responder's identity is verified using public-key cryptography. First, the Initiator sends a **DPP Authentication Request** message that includes its public bootstrapping key to ensure the Responder can identify the Initiator and a randomly generated number that is used only once (this number is referred to as a nonce). The nonce is encrypted using the Responder's public bootstrapping key, which was set prior to the DPP using an out-of-band channel (see Section 4.4).

The Responder receives this message, decapsulates it, and decrypts the nonce using the private key. If mutual authentication is enabled, the Responder looks up the received public key in the list of public keys the Responder is configured to trust. If no match is found, the Responder will abort the DPP. If mutual authentication is disabled, the Responder will implicitly trust the public key received in the DPP Authentication Request message, assuming the DPP was never completed on the device (otherwise it will only trust the imprinted key from the previous instance of the DPP). Then, the Responder will

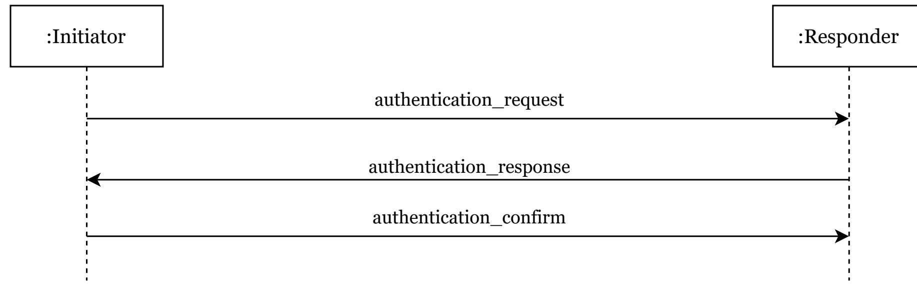


Figure 4.3: Messages transmitted during the Authentication phase

send a **DPP Authentication Response** containing a status code identifying whether the verification was successful and – if the verification was successful – the received nonce from Initiator as well a second nonce generated by the Responder, with both nonces encrypted by the Initiator's public bootstrapping key.

Using the nonces that both the Initiator and the Responder possess now and both bootstrapping keys, both devices are able to use the elliptic curve Diffie-Hellman algorithm to calculate a shared key that will be used to encrypt subsequent messages using the AES algorithm. Because the Diffie-Hellman algorithm is used, the shared key will be known only to the two devices. If the calculation is successful (this assumes the Initiator received back the same nonce it had sent to the Responder in the DPP Authentication Request), the Initiator sends a **DPP Authentication Confirm** message indicating the process was successful. This message contains data encrypted by the shared key. [33]

4.5.3 Configuration phase

In the configuration phase (shown in Figure 4.4), the connection details are generated by the Configurator and provided to the Enrollee. Unlike the previous two phases, the Configuration phase uses the *Enrollee* and the *Configurator* roles instead of the *Initiator* and the *Responder*. The configuration phase consists of three messages, the DPP Configuration Request, the DPP Configuration Response, and the DPP Configuration Result. The content of all three messages is encrypted using the shared key generated in the Authentication phase.

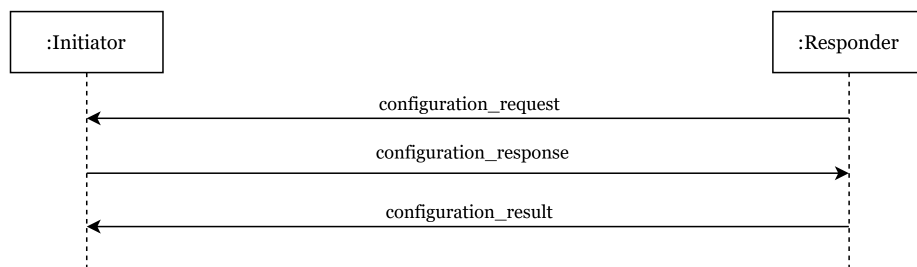


Figure 4.4: Messages transmitted during the Configuration phase

The **DPP Configuration Request** is sent by the Enrollee to the Configurator. The request contains configuration attributes that specify the kind of configuration details required and a new nonce that will be repeated in the following DPP Configuration messages to verify their integrity.

The **DPP Configuration Response** contains the configuration details assigned by the Configurator device, the nonce from the previous DPP Configuration Request message, and a success indicator value. The success indicator value is transmitted unencrypted, so the Enrollee may receive the error code if a wrong key was used to encrypt the message.

The structure of configuration details in the DPP Configuration Response message varies based on whether the Enrollee is a station or an access point. *STA configuration* contains the SSID name of the network to connect to, connectivity policy settings, and a special Connector object that can be validated by an AP or a WPA2-Personal passphrase or similar method[1] for legacy devices. *AP configuration* contains the SSID name, the specific content of some fields that should be transmitted by the AP, the operating channel or band information, and the Connector object or legacy credential details.

The configuration session is ended by the **DPP Configuration Result** message, which is sent by the Enrollee. This message informs the Configuration whether the Enrollee was able to receive and apply the configuration details. The message consists of a status code and the nonce from previous messages, with both values encrypted using the shared key. [33]

4.5.4 Access phase

The access phase (shown in Figure 4.5) is not a part of the provisioning process, as it needs to be executed every time the station connects to a Wi-Fi network. While the phase is illustrated as a single *„wifi connect“* message in Figure 4.2, it actually consists of several 802.11 messages and corresponds to the standard process when a wireless device connects to a Wi-Fi network.

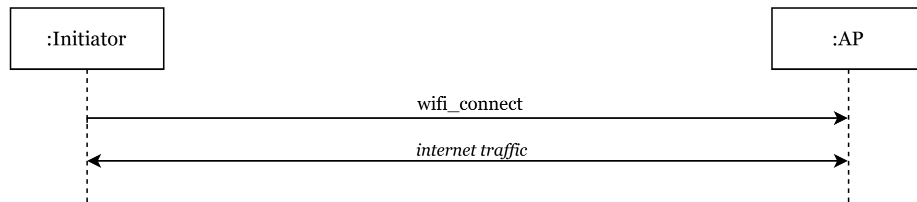


Figure 4.5: Messages transmitted during the Access phase

Unless the STA uses a legacy authentication method to connect to a Wi-Fi network, Peer Discovery Request and Peer Discovery Response frames are used to verify that both STA and AP received their Connector object from the same Configurator, or by two different Configurators that trust each other. This allows for the AP to be configured after some of the stations were already provisioned. However, as very few access points support this protocol at the moment, it will not be considered in this thesis. Once the devices are verified, standard IEEE 802.11 procedures will be followed to connect to the AP. [33]

[1] The DPP accepts the following legacy authentication procedures: WPA2-Enterprise, WPA2-Personal, WPA3-Enterprise, and WPA3-Personal with either X.509 certificate, PSK, PSK Passphrase, or SAE password credentials. [33]

Chapter 5

ESP-32

This chapter introduces the **ESP32-S2-Saola-1** device that was chosen for the implementation of the proposed zero-touch provisioning protocol and its features.

This chapter first overviews the device as a whole in Section 5.1. Then, sections 5.2 and 5.3 introduce the development toolkit and build system used to develop apps for the device. Section 5.4 presents the toolkit's menuconfig tool that can be used to provide the application with user-configurable options that are passed into the application code at compile time. Next, Section 5.5 overviews the ESP-NOW protocol that provides connectionless device-to-device communication. This is followed by Section 5.6 that introduces the cryptographic library used in the implementation. After this, Section 5.7 presents the eFuse memory that can be used to store encryption keys and Section 5.8 introduces the flash encryption and secure boot features. Then, Section 5.9 introduces the non-volatile storage that can store data between the device's power cycles. And finally, Section 5.10 introduces selected features provided by the ESP32's underlying FreeRTOS operating system, namely timers, tasks, and queues.

5.1 Basic features

The ESP32-S2 microchip (shown in Figure 5.1) is a low-power Systems-On-Chip (SoC) microchip that features a single-core 32-bit microprocessor that can run at frequencies up to 240 MHz. The chip can store data in 128 kB ROM, 320 MB RAM, and up to 4 MB flash memory. The chip provides similar interfaces as any common microchip device, including Wi-Fi, GPIO, UART, SPI, and I^2C. The chip also contains security acceleration modules that enable secure storage of private and public key pair, secure boot, flash encryption, random number generator, and acceleration for some cryptographic operations. [13]

The S2-Saola-1 variant of the ESP32 also includes a development board. The board features a USB-to-UART bridge, allowing easy development of the device through the USB port of a development computer. The board also contains a boot button, a reset button, a programable RGB LED, a voltage indicator LED, and input/output pins. [14]

5.2 Development toolkit

The manufacturer provides a development toolkit that supports an automatic build and deployment of applications. To develop an app, one needs a computer running Windows, Linux, or Mac OS with Python installed, the ESP32 device itself, and a micro-USB to

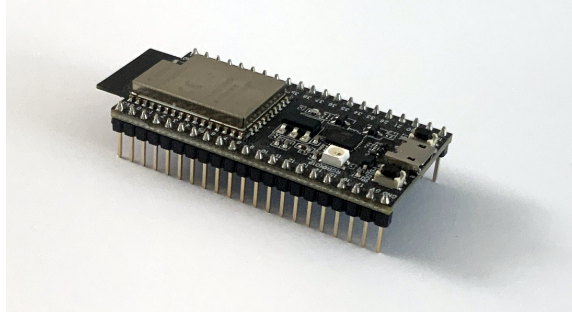


Figure 5.1: The ESP32-S2-Saola-1 microchip

USB cable to connect them. Then, the development toolkit must be downloaded from the manufacturer's website and installed according to the instructions (both available at [17]). The installation of the toolkit is straightforward, as it only requires running the provided batch or shell file (depending on the computer's operating system). [17]

The ESP32 microchip is programmed using applications written either in C or C++ that are executed by a real-time FreeRTOS operating system. Next to the standard C/C++ library functions, the development toolkit provides an API with functions that provide access to the chip's hardware modules, as well as high-level functions that provide commonly used software functionality. [17]

A typical development process is as follows: first, a boilerplate project is created using the toolkit or by copying an existing project. Then, the source code can be modified as desired. Before execution, the chip with the development board is connected to a computer using USB, and the correct USB port is selected in the toolkit if more than one development board is connected. The project is then configured, build, and flashed to the device using appropriate toolkit commands as shown in Section 5.3. Finally, a monitor that displays the standard output of the device can be opened. [17]

5.3 Build system

The development toolkit provides commands to build, flash, and monitor the application. To use this toolkit, it must be first initialized using the `get_idf` command. This command initializes the required environment variables and once run, the development toolkit can be used until the terminal window is closed. In a new shell (in a new terminal window), the `get_idf` command needs to be executed again. As the command takes several seconds to execute, it is not advisable to run it automatically when a new shell is created, for example in the `.bashrc` file.

Once the toolkit is initialized, the user needs to navigate to the project's directory in the terminal (using the command `cd`) and set up the target environment device to `esp32s2` (if an ESP32-S2 device is used) using the command in Listing 1. This command only needs to be executed once per project's lifetime. [17]

```
idf.py set-target esp32s2
```

Listing 1: Command to set the target device to ESP32-S2

```
idf.py build  # build project
idf.py -p /dev/ttyUSB0 flash  # flash project to device /dev/ttyUSB0
idf.py -p /dev/ttyUSB0 monitor  # monitor device /dev/ttyUSB0
```

Listing 2: Commands to build, flash, and monitor the project

The commands listed in Listing 2 can be used to build the project, flash it to the ESP32 device, and monitor the output of the device respectively. In order to flash the device and to monitor it, the port to which the device is connected must be provided. The port used in this example is `/dev/ttyUSB0`. After executing the last command, use the keyboard shortcut `ctrl + ]` to exit the device monitor. Alternatively, the command in Listing 3 can be used to perform all three actions (build, flash, and monitor) at once. [17]

```
idf.py -p /dev/ttyUSB0 flash monitor
```

Listing 3: A single command to build, flash, and monitor the project

Finally, either of the commands shown in Listing 4 can be used to delete the files that were generated when the project was built. The first command deletes the output from the C-compiler only, forcing a full rebuild the next time the project is built. The second commands delete all auto-generated files, including those generated by CMake. [17]

```
idf.py clean  # deletes the C-compiler generated files
idf.py fullclean  # deletes all auto-generated files
```

Listing 4: Commands to delete the build output files

5.4 Menuconfig

Many computer applications use some kind of text file to configure their behavior in a specific installation. For ESP32 applications, the development toolkit features a menuconfig tool that provides pre-defined configurable variables into the application code at build time. To use this feature, all configurable variables need to be defined first in the `Kconfig.projbuild` file. An example of such definition for one variable is shown in Listing 5. [19]

```
config IOT_PROV_CHANNEL
    int "Wi-Fi channel"
    default 1
    range 0 14
    help
        The Wi-Fi channel to send and receive ESP-NOW messages
```

Listing 5: A definition of a configurable variable to choose the Wi-Fi channel. The variable can be set to an integer ranging from 0 to 14, with the default value of 1.

Once this file is created, the application's user can use the `idf.py menuconfig` command to set the variables' values for the specific installation. This command provides an easy-to-use command-line interface to set the value of each variable, as displayed in Figure 5.2. [19]

Figure 5.2: Menuconfig interface

From the developer's perspective, all values defined using the menuconfig tool are automatically accessible via constants in the application code. For example, the value from Listing 5 can be accessed anywhere in the application code by reading the constant `CONFIG_IOT_PROV_CHANNEL`. [19]

5.5 ESP-NOW

ESP-NOW is a proprietary protocol that allows connectionless communication between ESP32 devices, meaning the devices do not need to be connected to an Access Point in order to communicate with each other [11]. Because of this, the ESP-NOW protocol will be used in the proposed protocol during the provisioning of devices.

To offer the ability to communicate without being connected to an AP, the ESP-NOW protocol leverages the IEEE 802.11 Management Frames. These frames are sent by network devices on various occasions, often to devices that are not connected to any network. A well-known example of a Management frame is the Beacon frame, which is periodically sent by most Access Points to announce their SSID to devices that want to connect to their network. The ESP-NOW protocol uses a different Management frame – an Action frame – because its content may be filled with any vendor-specific data and it can be sent and received by unconnected devices. [22]

Thus, the ESP-NOW messages are implemented as vendor-specific data in the Management Action Frame. Alongside the vendor-specific data field, the Action Frame also contains fields for the source and destination MAC addresses, which will be used to identify the source and destination device, thus the MAC addresses do not have to be present in the vendor-specific data field. The vendor-specific data field of an ESP-NOW message consists of the length of the data sent over ESP-NOW, various identifiers[1] indicating this is a vendor-specific element containing an ESP-NOW message in the current version of the protocol, and the application-layer data. [11]

The application-layer data may contain arbitrary binary data, however, the ESP-NOW protocol limits the maximal length of the application-layer data to 250 bytes [11]. As will be shown in Section 7.4, this will have to be circumvented by a custom intermediate protocol providing message fragmentation, because there will be a need to send messages longer than 250 bytes over the ESP-NOW protocol.

[1]Namely, the fields and their values are: Element ID *(221: vendor-specific element)*, Organization Identifier *(0x18fe34: Espressif)*, Type *(4: ESP-NOW)*, and Version *(might vary)*

The ESP-NOW protocol supports sending both unicast and broadcast messages. For unicast messages, the destination is identified by the target's MAC address. [11]

The protocol also supports message encryption using a 128-bit AES shared key. A message will be correctly sent and received only if both the sender and receiver are configured with the same shared key. However, this key must be pre-shared between the devices using an out-of-band channel, because the protocol does not offer any pre-build way to negotiate the shared key using the ESP-NOW protocol, such as the Diffie-Hellman algorithm. Because of this, the protocol's encryption feature will be disabled in this thesis and all messages will be encrypted in the application layer as needed. [11]

5.6 Cryptographic functions

The ESP32 device features a hardware module for the acceleration of cryptography-related tasks. In particular, this includes the AES Accelerator, the SHA Accelerator, the RSA Accelerator, the Random Number Generator, the Digital Signature, and the HMAC Accelerator modules [15]. These modules can be handled manually by performing read/write operations to the correct registers at a specified order and timing, but the development toolkit also includes an ESP-specific implementation of the mbedtls[2] library [12]. This library implements all common cryptographic operations, such as the parsing of keys in the PEM format, random number generation, data encryption and decryption, hashing, and more. The library uses the hardware acceleration modules internally as needed.

5.7 eFuse

To prevent the private keys for asymmetrical cryptography and the shared encryption keys for symmetrical cryptography from being leaked from the device, the ESP32 features an eFuse storage. The eFuse storage consists of 11 independent blocks indexed from 0 to 10. Each block is one-time programmable per the microchip's lifetime, meaning that once a value is written to an eFuse block, it can never be modified or erased. Some blocks (block 0 and block 1) are divided into several parameters, where each parameter can be programmed individually. [15]

When an eFuse block is programmed, the following parameters need to be provided:

- the **block number** to be programmed (0 through 10),
- the name of the **parameter** to be programmed, if only one parameter is programmed rather than the full block (applies only to blocks 0 and 1),
- the **value** to be stored (256 bits of data if the full block is programmed),
- the **read protection bit**, indicating whether the stored value should be accessible using software (either *true* or *false*; applies only to blocks 4 through 10), and
- the **key purpose**, storing the purpose of the saved key (applies only to blocks 4 through 9) [15].

The values of the read protection bits and key purposes are stored in the eFuse block 0, thus those values cannot be modified once written as well. Each of the 11 eFuse blocks is

[2] https://tls.mbed.org/

intended for a different use, thus certain data may be stored only in selected eFuse blocks. Table 5.1 shows what type of data can be stored in individual eFuse blocks. [15]

Block number	Usage
0 – 2	System data
3	Arbitrary user data
4 – 9	Encryption keys or user data
10	System data

Table 5.1: Parameters to store in individual eFuses blocks [15]

As the eFuse can prevent some values from being software accessible, it is recommended to use this module to store device-specific secret keys. If a value is set as software inaccessible, it can still be accessed by other hardware modules, such as the Digital Signature module. For encryption keys (in blocks 4 through 9) the *key purpose* value further limits which hardware modules are permitted to read the key. However, some modules such as the AES Accelerator do not seem to support the usage of keys stored in an eFuse at the moment. [15]

As one eFuse block is capable of storing up to 256 bits of data, it is not possible to store a private RSA key (which is usually 2048 or 4096 bits long) into an eFuse. To protect such keys, the following strategy is used: first, the private key is encrypted using a randomly generated AES key. Then, the AES key is saved in the software-inaccessible eFuse and the encrypted RSA key is stored in unprotected flash storage. To sign or decrypt messages using this private key, a dedicated hardware module that is permitted to read the AES key from the eFuse is used, while the encrypted private key is provided to the module by the application software along with the data to be signed or decrypted. [15]

5.8 Flash encryption and secure boot

The ESP32 microchip also offers features to prevent unauthorized access to data stored in flash memory. This flash memory stores the compiled program code, but it may be also used by the application to store user data, as will be shown in Section 5.9. The protection features covered in this section are flash encryption and secure boot. Both features prevent unauthorized operation with flash memory if an attacker gains physical access to the device.

The **flash encryption** prevents the attacker from reading the data stored in the internal flash memory if they gain physical access to the device. As the flash memory is connected to the chip's CPU using the SPI (Serial Peripheral Interface), it may be possible for an attacker with physical access to the device to read out the content of the flash memory, thus gaining access to the assembled application code and all data the application stores in the flash memory. By encrypting the data, the attacker will not be able to interpret the data collected from the flash memory. The flash encryption will also prevent the attacker from intentionally changing the data (to a known value), but it will not stop the attacker from changing the data to random values, because an attacker is able to write to the flash memory but they have no control about how the data will change during decryption. The encryption key is generated automatically when the flash encryption is enabled and stored in a software-inaccessible eFuse. [16]

The **secure boot** prevents an attacker from unauthorized modifications of the application code, for example by flashing a completely new application to the device. This is

achieved by signing the application using a private RSA key before flashing and forcing the IoT device to execute only applications that were signed by a particular key. The key need for the signature verification is stored in an eFuse when secure boot is enabled, and therefore cannot be changed once set. [20]

The steps needed to enable flash encryption are given in [16], while the steps to enable secure boot are given in [20].

5.9 Non-volatile storage

As was indicated in the previous section, the ESP32 uses a flash memory to store application and user data. Depending on the variant of the chip, the size of this memory ranges from 2 MB to 4 MB [13]. It is possible to store user data in this memory, which is useful because data in this memory are not erased when the device is disconnected from the power supply. This section shows two possible approaches to storing user data in flash memory.

The *static file* approach can be used when the application needs to refer to a file that is available at the time of compilation and the file is not changed by the application. In this approach, the file is first registered in `main/CMakeLists.txt` using the `EMBED_FILES` as shown in Listing 6. [10]

```
idf_component_register(
    SRCS "my_app.c"
    INCLUDE_DIRS ""
    EMBED_TXTFILES "public_key.pem"
)
```

Listing 6: Example of a registration of a static file in CMaleLists

In the application code, the file can be accessed as a constant external variable as shown in Listing 7. The `file_start` variable is a pointer to the beginning of the file, and the `file_end` is a pointer to the end of the file. [10]

```
extern const uint8_t file_start[] asm("_binary_public_key_pem_start");
extern const uint8_t file_end[]   asm("_binary_public_key_pem_end");
```

Listing 7: Accessing a static file in the application code

The second approach is to use the *Non-volatile storage (NVS) library*. This library allows read and write access to a designed part of the flash memory, where each value stored in the NVS is identified by a developer-chosen identifier, thus the NVS acts as key-value storage. [18]

The storage first needs to be initialized using the `nvs_flash_init` and `nvs_open` functions. Then it is possible to read and write string values in the storage using the `nvs_get_str` and `nvs_set_str` functions. The NVS library also offers functions to store data types other than string, such as `nvs_get_u8`, `nvs_get_u16`, and more. [18]

5.10 FreeRTOS features

The ESP32 microchip is powered by a port of the FreeRTOS[3] operating system. Because of this, the developer is free to use the API provided by this system. This section focuses on crucial features provided by the FreeRTOS that are used in almost every ESP32 application.

5.10.1 Timers

The most basic FreeRTOS feature covered in this thesis is the timer. A timer is used to schedule an action that should be executed after a given period of time. The action can be either executed once or periodically until canceled. [1]

To use the timer, it must be first initialized using the `xTimerCreate` function. This function initializes the timer but it does not start it yet. During the initialization, the timer's period and the callback function are set, as well as the flag whether the timer should be fired once or periodically. The `xTimerStart`, `xTimerStop`, `xTimerReset`, and `xTimerDelete` functions can then be used to start, stop, reset, and delete the timer respectively. The application is free to execute other code while the timer is running. An interrupt will be issued when the timer expires. [1]

It is possible to use the timer to check for the counterpart's inactivity, for example, to reset the provisioning process if no message is received from the other device for a given period of time. This is done by starting a timer when a request is sent from the device. When the reply arrives, the timer is stopped (if no more messages are expected) or reset (if additional messages are expected from the counterpart), as illustrated by the *Inactivity timer 1* in Figure 5.3. If the timer's callback function is triggered, it means the counterpart is inactive as shown by the *Inactivity timer 2* in Figure 5.3.

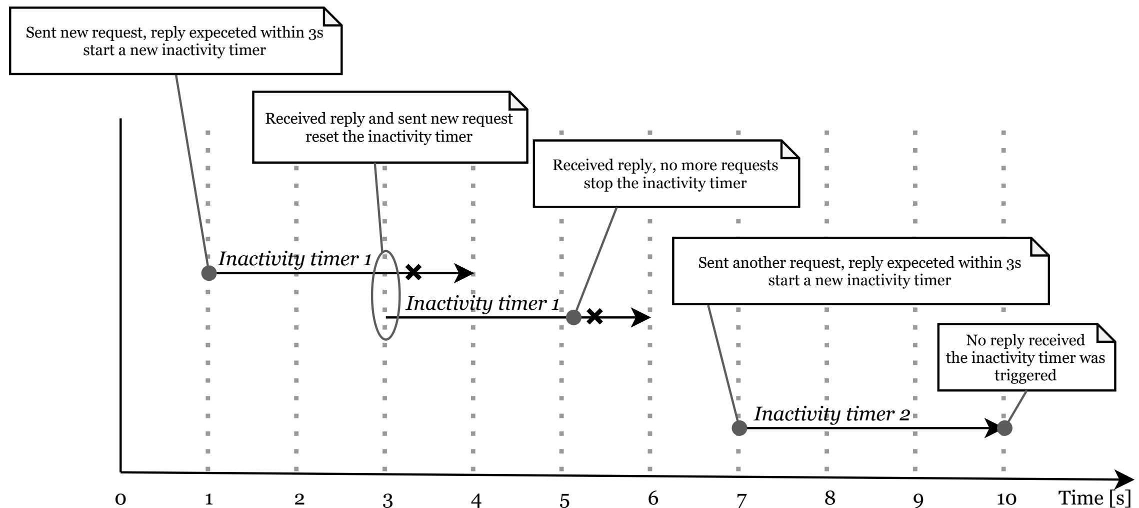


Figure 5.3: An example of using timers to track requests that did not receive a reply

5.10.2 Tasks

The FreeRTOS tasks are similar to threads in common programming languages. However, as the ESP32 chip used in this thesis has only one core, it will not be capable of running

[3] https://www.freertos.org/

multiple tasks simultaneously. The main advantage of using tasks on single-core processors is that it is possible to set a different priority for each task, thus allowing one task to interrupt a task with lower priority. [2]

A typical example of task usage is to create a high-priority task for storing incoming packets from the network interface into a buffer. This task is idle most of the time, but when the network interface receives a packet, it is essential to store the incoming packet as soon as possible, so the network interface is ready to receive more packets. This is possible because the operating system will switch to this task when a packet is received because of the task's high priority, while the operating system will execute other tasks when the high-priority task is idle. [2]

A task is implemented by a function that never returns as shown in Listing 8. It is possible to pass a user-defined pointer into this function using the pvParameters parameter. [2]

```
void myTaskFunction(void *pvParameters) {
    while (true) {
        // Task code
    }
}
```

Listing 8: Implementation of a task

The FreeRTOS API includes the xTaskCreate and vTaskDelete functions to create and delete tasks. When creating a task, a pointer to the task's entry function and the value to pass into pvParameters must be given, as well as the the task's priority and the size of the stack used by the task. It is possible for a task to call vTaskDelete on itself if it wants to be deleted. There will be an error that leads to a device reboot if the task's entry function returns and the task is not deleted. [2]

5.10.3 Queues

A queue is a data structure that allows to store a collection of elements that can be accessed using the FIFO (first-in-first-out) paradigm. In FreeRTOS applications, queues are typically used to pass data into a task or out of it, because a single queue can be shared across multiple tasks. Typically, a queue is shared across two tasks, where one task writes data into the queue and the other one reads data out of it. [3]

The xQueueCreate function is used to initialize a queue. During initialization, the size of one element and the maximal capacity of the queue must be specified. The queue will not reallocate if it is full, and thus the operation to add an element to it might fail if the queue is full. [3]

The xQueueSend and xQueueReceive functions are used to add an element to a queue and to receive an element from the queue respectively. The xQueueReceive function will automatically remove the element that was received from the queue. Alternatively, the xQueuePeek function can be used to read the element from the front of the queue without removing it. All three functions may block the thread that has called them if the queue is empty (for xQueueReceive and xQueuePeek) or full (for xQueueSend). Because of this, all three functions contain an xTicksToWait parameter that sets the maximum amount of

time[4] the function is willing the wait. If the value is set to 0, the function will never block the task. The function will return an error code if the value could not be received or written within the interval. [3]

The `vQueueDelete` function deletes the queue. [3]

[4] The time is specified in the number of CPU ticks, but the `portTICK_RATE_MS` constant that contains the interval of CPU ticks in milliseconds may be used to convert from microseconds to CPU ticks

Chapter 6

Analysis

As shown in Chapter 3, the current solutions for network provisioning used in current IoT devices are hard to use – especially when setting up a large number of IoT devices – and often are not very secure. This could be – and in the future will be – overcome by switching to the Wi-Fi Easy Connect standard as shown in Chapter 4, however, this standard is in the draft stage and no devices sold today support it yet.

The upcoming standard aims to support all network provisioning needs of all devices in the future, as suggested by the various configuration options the standard provides, some of which were shown in Chapter 4. For example, depending on the capabilities of devices in a given scenario, either the Enrollee or the Configurator may start the provisioning process. For example, if a smart water heater is about to join a Wi-Fi network, the DPP would be most likely started the Configurator by scanning a barcode on the water heater, as most water heaters have no user interface. On the other hand, if a guest in a hotel would like to connect to the hotel's Wi-Fi network, the DPP would be started by an Enrollee, as the guest would scan the QR code received from the hotel staff on their own STA device that is about to join the network.

Another example of the complexity of the upcoming standard is the fact that one network can be configured using multiple Configurator devices. Thus, a feature for delegation of capabilities from one Configurator to another that was not described in the previous chapter must be defined. Also, the protocol supports all Wi-Fi authentication protocols ranging from WPA-Personal with Pre-Shared keys to WPA2-Enterprise with client X.509 certificates, including a new protocol using a Connector object introduced by this standard.

Thus, rather than implementing the new Wi-Fi Easy Connect standard in full, this thesis aims to design and implement a new lightweight provisioning protocol that demonstrates the capabilities of the concepts introduced in this upcoming standard for network provisioning of IoT devices. The proposed protocol will use the same four phases as the Wi-Fi Easy Connect standard (Bootstrapping, Authentication, Configuration, and Access), but the messages sent in individual phases will be modified to reduce the complexity of the protocol, with regards to the desired use case of provisioning IoT devices (for example, the proposed protocol will not be able to provision an access points).

The proposed protocol should follow the findings in Section 3.7, namely, it should be secure, require minimal user-interaction, and consider interoperability with different protocols.

Chapter 7

Design

This section explains the design of the implemented provisioning protocol. As defined in Chapter 2 and observed in many current implementations in sections 3.3, 3.4, 3.5, and 3.6, as well as in the upcoming Wi-Fi Easy Connect standard (in Section 4.1), the protocol will use two devices for the provisioning process – an Enrollee, and a Configurator.

The Enrollee is the new unconfigured device that lacks the network credentials needed to connect to an existing Wi-Fi network. The Configurator, on the other hand, is a device that stores the network credentials of all IoT devices and provides them to Enrollees upon request. For simplicity, this device will be also instrumented by an ESP32 microchip. Thus, there will be $N + 1$ devices necessary to provision N Enrollees (one extra ESP32 microchip is needed to act as the Configurator).

Figure 7.1 highlights the steps that need to occur during the provisioning process (a diagram showing all messages that are transmitted between the Enrollee and the Configurator will be shown later in Section 7.5). All messages will be transmitted using the connectionless protocol ESP-NOW introduced in Section 5.5, as the Enrollee has no other means of internet connectivity yet.

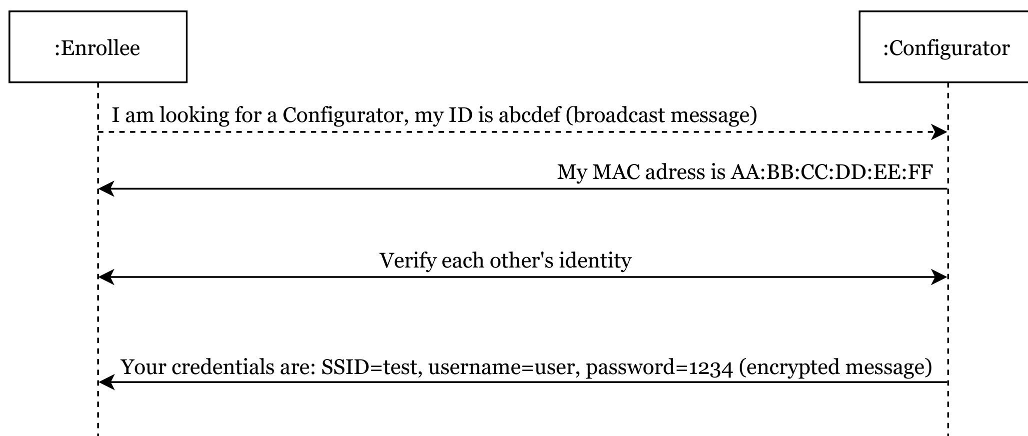


Figure 7.1: Overview of messages sent during the provisioning process

First, the Enrollee needs to locate the Configurator by sending a broadcast message. This message includes an Enrollee's identifier, which is represented by a hash of the Enrollee's public key that will be generated in Section 7.2. If a Configurator receives this message, it will attempt to look up the received identifier in a list of known identifiers. The presence of the Enrollee's identifier in a list maintained by the Configurator is not sufficient

to authenticate the Enrollee, but if the Configurator does not find the Enrollee's identifier, the Configurator will not engage in the provisioning protocol any further. This way, the Enrollee will not be burdened to negotiate with Configurators that do not trust the Enrollee if the Enrollee is within the reach of multiple Configurators.

Second, the devices need to verify each other's identity. The provisioning protocol supports two operating modes of authentication – either mutual or non-mutual. In both modes, the Configurator will verify Enrollee's identity. In the mutual mode, the Enrollee will also verify the Configurator's identity to prevent the Enrollee from receiving credentials from an imposter Configurator. The mutual mode, however, requires additional configuration steps, thus making it less user-friendly.

And third, after the devices' identities are verified, the Configurator will send the network credentials to the Enrollee. To prevent an attacker from sniffing the credentials while they are sent to the Enrollee, the credentials will be encrypted by a private key only known to the Enrollee.

The rest of the chapter is structured as follows. First, Section 7.1 gives details on the authentication mechanism used in the provisioning protocol. Then, sections 7.2 and 7.3 list the prerequisites that the Enrollee and Configurator must satisfy before the provisioning protocol can be started. This is followed by Section 7.4, which introduces an intermediate fragmentation protocol that is used to transfer messages that are too long to be transmitted over ESP-NOW. And finally, Section 7.5 introduces the structure of all messages that are used by the provisioning protocol.

The proposed protocol will be later evaluated from a user experience point of view in Section 9.2 and from a security point of view in Section 9.3.

7.1 Authentication of devices

The provisioning protocol uses public-key cryptography to build trust between devices. Each device (both Enrollees and Configurators) contains a unique pair of private and public RSA key, that needs to be generated before the provisioning protocol is initiated. The private keys will never leave the devices that they were generated for. The public keys may be transmitted to other devices as needed.

Regardless of whether the mutual authentication is enabled, the Configurator needs to contain public keys of all Enrollees that are allowed to use the services of the Configurator. However, as the storage abilities of IoT devices are limited, the Configurator will only store SHA-256 hashes of the Enrollees' public keys. When the provisioning process is started, the Configurator will request the full public key from the Enrollee. The authenticity of the received public key will be verified by a recalculation of its hash on the Configurator's side.

If the mutual authentication is enabled, the Enrollee will also need to contain the public key of the Configurator. Because the Enrollee only needs to know the public key of one Configurator, the full public key will be stored in the Enrollee rather than its hash. This will reduce the amount of data transmitted between devices during the provisioning process.

The procedure taken to authenticate the devices depends on whether the mutual authentication is enabled. If it is enabled, the challenge-response algorithm shown in Figure 7.2 is used to verify the parties. The *pk_en* and *pk_conf* labels in Figure 7.2 indicate that the part of the message in the preceding square brackets (between *[* and *]*) was encrypted using the Enrollee's (in case of *pk_en*) or Configurator's (in case of *pk_conf*) public key.

The algorithm starts by generating a random number *nonce1* on Configurator. The generated number must be large enough to ensure that the same number will never be

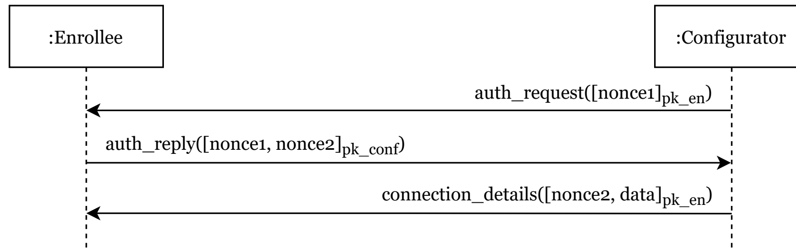


Figure 7.2: The authentication procedure used if the mutual authentication is enabled

generated again in the device's lifetime, otherwise, it would be possible to perform a replay attack. The generated number is then encrypted using the Enrollee's public key and sent to the Enrollee using the **`auth_request`** message.

The Enrollee decrypts the number *nonce1*, and generates its own random number *nonce2*. After that, the Enrollee constructs the **`auth_reply`** message that contains both nonces encrypted by the Configurator's public key. Because both devices knew each other's public keys beforehand and only the rightful devices possess the corresponding private keys, the Configurator can be certain the `auth_reply` message was sent by the Enrollee if the received *nonce1* matches the original, as only the Enrollee was able to decrypt the `auth_request` message.

As the identity of the Enrollee is verified now, the Configurator is free to send the **`connection_details`** message to the Enrollee. This message contains the Enrollee's network credentials as well as the *nonce2* that was sent by the Enrollee in the `auth_reply`, with both values encrypted by the Enrollee's public key. The Enrollee is then able to verify the received *nonce2*, and if does not match the original value, it can simply ignore the message.

Note: because all encrypted messages transferred in this protocol are relatively small (less than 1 kilobyte), the messages are encrypted using RSA rather than negotiating a shared AES key, which is faster for encrypting a large amount of data.

If the mutual authentication is disabled, the Enrollee will not verify the identity of the Configurator. Because of this, the `auth_request` and `auth_reply` messages will not be transmitted, and the Configurator will send the `connection_details` message containing the Wi-Fi credentials to the Enrollee, which will be encrypted by the Enrollee's public key. Because the Configurator knows the Enrollee's public key beforehand and only the Enrollee holds the corresponding private key, no other device than the Enrollee will be able to decrypt the message.

7.2 Enrollee's prerequisites

As was mentioned in the previous section, the Enrollee needs to generate a unique pair of RSA keys before the device can engage in the provisioning protocol. If the mutual authentication is disabled, no other configuration needs to be done on the Enrollee. If the mutual authentication is enabled, the Configurator's public key must be stored in the Enrollee device too. Note that in the mutual authentication mode, the Enrollee can only be configured to trust one Configurator.

7.3 Configurator's prerequisites

The Configurator device also needs to have generated a unique pair of RSA keys, just like the Enrollee devices. Next to this, the Configurator also needs to be configured with a list of the connection details that should be provided to Enrollees. This list contains one entry per authorized Enrollee, where each entry contains the following values:

- SHA-256 hash of the Enrollee's public key,
- SSID of the network the Enrollee should connect to,
- the network's username (empty if the network uses a PSK authentication only), and
- the network's password.

Instructions on how to store this data in the ESP32 device are given in Section 8.7.

7.4 Message fragmentation

The ESP-NOW protocol is capable of transmitting messages that are up to 250 bytes long. However, as the Configurator only stores hashes of the Enrollees' public keys to save space, the Enrollee needs to be able to send the full public key to the Configurator upon request. As RSA keys are typically 2048 bits (256 bytes) or 4096 bits (512 bytes) longs (with the effective size even larger because of the PEM encoding), it is crucial to be able to transmit messages longer than 250 bytes.

This is achieved by supporting message fragmentation on the application level. The provisioning protocol recognizes both non-fragmented and fragmented (multipart) messages. If a message is short enough (250 bytes or less) to be sent without fragmentation, it will be sent as-is to reduce overhead.

If a message is longer than 250 bytes, it will be split into several 245-bytes-long fragments (the last fragment may be shorter so the lengths of all fragments added together match the length of the original message). Then, each fragment is encapsulated into the structure shown in Table 7.1 and sent over ESP-NOW.

Type	Field	Offset	Size	Description
`uint8`	`magic`	0 B	1 B	A magic number to identify a multipart message (always `0x11`)
`uint8`	`msg_id`	1 B	1 B	A message ID. This ID is shared by all fragments of the same original message.
`uint16`	`msg_size`	2 B	2 B	The size of the full message in bytes
`uint8`	`part_num`	4 B	1 B	A number of the fragment within the message
`uint8[]`	`data`	5 B	$\leq$ 245 B	Fragment data

Table 7.1: The structure of a fragment of a message

The **magic number** is used to identify a multipart message (all non-fragmented messages are guaranteed not to start with this number). The **message ID** is used to identify the original message the fragment belongs to if multiple multipart messages are in transit (the message is identified on the receiver by the message ID and the sender's MAC address).

The **message size** contains the size of the original message (before fragmentation). This number can be used to calculate the number of parts and the size of each fragment because all-but-last fragments are guaranteed to contain 245 bytes of the original message. The **part number** identifies the order of fragments if the messages are delivered out-of-order (the number of the first fragment within a message is 0). And finally, the **data** field contains the fragment's application data.

The protocol does not acknowledge the reception of individual fragments to the sender, as this is done by the ESP-NOW protocol internally. The maximum size of the original message that this protocol is capable of fragmenting is 64 kB. This limitation is caused by the size of the `msg_size` field, however, the implementation of this protocol will decrease this limit further.

7.5 Messages

This section defines all messages that are sent over ESP-NOW in this provided protocol. A sequence diagram illustrating the flow of these messages is shown in Figure 7.3.

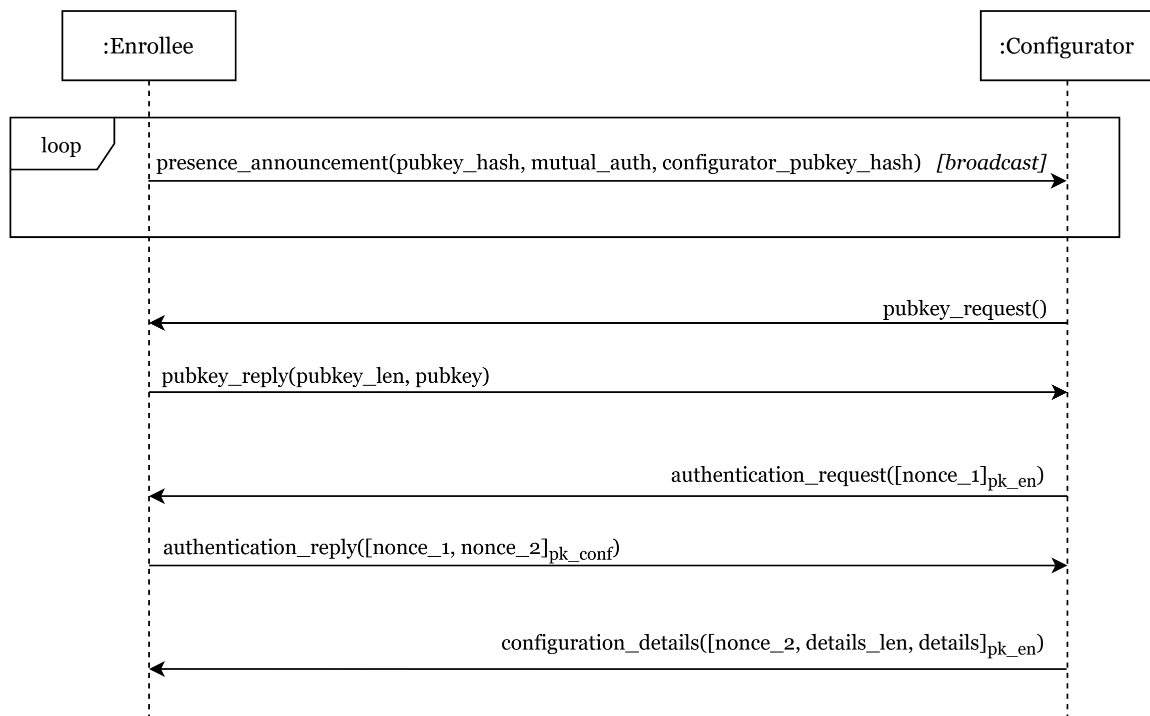


Figure 7.3: A sequence diagram of messages transmitted during the provisioning protocol with mutual authentication enabled

The provisioning protocol is started by the Enrollee, which periodically broadcasts the *Presence announcement* message until it receives a *Public key request* from a Configurator. The Enrollee replies to this request using the *Public key reply*, which contains the Enrollee's public key. If mutual authentication is enabled, both devices will exchange the *Authentication request* and *Authentication reply* messages to verify each other's identity. And finally, the configuration details are sent in the encrypted *Connection details* message.

To disambiguate between various types of messages, all messages share the following structure: the first 10 bytes of every message are reserved for the *message type* field, which

contains a pre-defined value for each type of message, e.g. presence for the presence announcement message. The value contains printable ASCII characters only, and the remaining bytes are set to zero (0x00) if the value is shorter than 10 characters. The message type is not encrypted during transfer. The following bytes of the message are defined by the specific type of message.

7.5.1 Presence announcement

The structure of the Presence announcement message is shown in Table 7.2. This message is periodically broadcasted by the Enrollee until a Configurator device is found.

Type	Field	Offset	Size	Description
uint8[]	msg_type	0 B	10 B	Type of message (always presence)
uint8[]	sta_pk_hash	10 B	32 B	SHA-256 hash of the Enrollee's public key
bool	mutual_auth	42 B	1 B	Whether the station requires the mutual authentication (true or false)
uint8[]	conf_pk_hash	43 B	0/32 B	SHA-256 hash of the Configurator's public key (present if mutual_auth is true)

Table 7.2: Structure of the Presence announcement message

The sta_pk_hash field contains an SHA-256 hash of the sending station's public key. This is followed by the mutual_auth flag that indicates whether the station requires mutual authentication. The value of this flag should be set to zero if the mutual authentication is disabled or to any non-zero value if the mutual authentication is enabled. If the mutual authentication is enabled, the presence announcement shall also contain the conf_pk_hash field that contains an SHA-256 hash of the Configuration's public key. This will allow any other Configurators in the area to ignore this message.

The length of this message must be either 43 bytes if mutual_auth is zero or 75 bytes if mutual_auth is non-zero. Note that the values of the fields in this message do not change over the Enrollee's lifetime, so the Enrollee can prepare this message once and rebroadcast it as many times as needed.

7.5.2 Public key request

When a Configurator receives a presence announcement message from an Enrollee, and the configurator is willing to engage in the provisioning protocol with this Enrollee, a Public key request message will be sent to the Enrollee. As can be seen in Table 7.3, this message does not carry any other data than the message type.

Type	Field	Offset	Size	Description
uint8[]	msg_type	0 B	10 B	Type of message (always pubkey_r)

Table 7.3: Structure of the Public key request message

However, because of the nature of the ESP-NOW protocol, the Enrollee will learn the Configurator's MAC address from the headers of the ESP-NOW protocol. The same approach is used by the Configurator to learn the Enrollee's MAC address from the received

presence announcement message, thus both devices are able to exchange unicast ESP-NOW messages from now on.

The Configurator will not respond to a Presence announcement message if any of the following conditions are satisfied:

- The Configurator's list of authorized enrollees does not contain the hash of the Enrollee's public key that was received in `sta_pk_hash`.
- The `conf_pk_hash` field is present and its value does not match hash of the Configurator's public key.
- The value of `mutual_auth` is zero and the Configurator is configured to require mutual authentication from all Enrollees.
- The message is malformed.

7.5.3 Public key reply

The Public key reply is sent by the Enrollee as a response to the Public key request message. The structure of this message is shown in Table 7.4.

Type	Field	Offset	Size	Description
`uint8[]`	`msg_type`	0 B	10 B	Type of message (always `pubkey`)
`uint16`	`pubkey_len`	10 B	2 B	Length of the public key (in bytes)
`uint8[]`	`pubkey`	12 B	any	The Enrollee's public key in PEM format

Table 7.4: Structure of the Public key reply message

The message contains the length of Enrollee's public RSA key and the public key itself. Because of the length of the RSA keys, this message is likely to be fragmented during transit, but this is opaque for the provisioning protocol.

The Configurator must verify that the public key received in this message matches the SHA-256 hash of this key that was received in the Presence announcement when it receives the Public key reply message.

7.5.4 Authentication request

If the mutual authentication is enabled, the Configurator will respond to the *Public key reply* message by an Authentication request message. The structure of this message is shown in Table 7.5.

Type	Field	Offset	Size	Description
`uint8[]`	`msg_type`	0 B	10 B	Type of message (always `auth_r`)
`uint8[]`	`nonce_1`	10 B	32 B	Random number used once

Table 7.5: Structure of the Authentication request message

This message contains a number that is used only once (nonce) which cannot be guessed by an attacker. To make sure an attacker cannot predict this number, a hardware random number generator is used to generate it. The number is 32 bytes (256 bits) long, which is the same as the length of the SHA-256 hash used to identify the Enrollee's public key,

thus the likelihood of this number being repeated during the device's lifetime is the same as finding an SHA-256 hash collision, assuming the random number generation module is reliable. The number is small enough that it does not need to be separated into multiple blocks in order to be encrypted by the RSA algorithm, thus the usage of a 256-bit number does not create any unnecessary overhead.

This message is encrypted by the Enrollee's public key before it is sent (note that the `msg_type` is not encrypted for debugging reasons, thus only the nonce will be encrypted).

7.5.5 Authentication reply

The Authentication reply message is sent by the Enrollee as a response to the Authentication request message. The structure of this message is shown in Table 7.6.

Type	Field	Offset	Size	Description
`uint8[]`	`msg_type`	0 B	10 B	Type of message (always `auth`)
`uint8[]`	`nonce_1`	10 B	32 B	The nonce from the Authentication request
`uint8[]`	`nonce_2`	42 B	32 B	Enrollee-generated number used once

Table 7.6: Structure of the Authentication reply message

In this message, the Enrollee repeats the nonce generated by the Configurator in the Authentication request to prove the Enrollee was able to decrypt it. Next, the Enrollee generates a second nonce that will act as a challenge for the Configurator. Both nonces will be encrypted by the Configurator's public key and the message is sent to the Configurator.

7.5.6 Connection details message

The last message that is sent in this provisioning protocol is the Connection details message, which is sent by the Configurator to the Enrollee. It is either sent as a response to the *Authentication reply* message if the mutual authentication is enabled or as a response to the *Public key reply* message if the mutual authentication is disabled. The structure of the message is the same in both cases, and it is shown in Table 7.7.

Type	Field	Offset	Size	Description
`uint8[]`	`msg_type`	0 B	10 B	Type of message (always `conn`)
`uint8[]`	`nonce_2`	10 B	32 B	The nonce from the Authentication reply
`uint16`	`connection_details_len`	42 B	2 B	Length of the connection details (in bytes)
`uint8[]`	`connection_details`	44 B	any	NULL-terminated connection details string

Table 7.7: Structure of the Connection details message

The field `nonce_2` repeats the nonce that the Configuration received from the Enrollee in the *Authentication reply* message to prove to the Enrollee that the Configurator was able to decrypt the *Authentication reply* message. If the mutual authentication is disabled, this value is filled will with NULL (`0x00`) bytes.

The `connection_details_len` field stores the length of the connection details string in bytes (including the trailing NULL byte). The `connection_details` string contains

the SSID name of the network and the Wi-Fi credentials in the following semicolon-separated format: `ssid;username;password`. If a network uses a pre-shared-key (PSK) authentication, which uses passwords only (rather than the combination of a username and password), the username is left empty, i.e. the connection string would be formatted as `ssid;;shared_key`.

The content of this message (except the `msg_type`) is encrypted using the Enrollee's public key to ensure no other device is able to decrypt the message.

Chapter 8

Implementation

This chapter overviews substantial details of the implementation. As both the Enrollee's and the Configurator's code share many modules in common, the code for both devices is implemented in a single project. A compile-time option is used to select whether the application should act as an Enrollee or a Configurator.

First, Section 8.1 describes how the RSA keys are generated and stored in the IoT devices. This is followed by Section 8.2, which provides details on how the Enrollees' Wi-Fi credentials are stored in the Configurator device. Then, sections 8.3 and 8.4 show how the wrapper modules for cryptography-related and network-related tasks were implemented. After this, sections 8.5 and 8.6 describe the modules that implement the Enrollee's and the Configurator's operation code. And finally, Section 8.7 describes how the application can be configured and executed from the user's perspective.

8.1 Key generation and storage

To soften the burden of having to generate a pair of RSA keys on the IoT device, extract the public key out of the device, and copy it to another IoT device (from an Enrollee to a Configurator or vice versa), this implementation generates the RSA keys outside the IoT device on a computer. On Linux, the keys may be generated using the `openssl` program as shown in Listing 9. On other platforms, similar tools to generate an RSA keypair may be used. The generated key should be at least 2048 bits long. The ESP32's cryptographic module supports up to 4096-bit keys.

```
# Generate a 2048-bit private key to private_key.pem
openssl genrsa -out private_key.pem 2048

# Extract the public key from private_key.pem and save it to public_key.pem
openssl rsa -in private_key.pem -pubout -out public_key.pem
```

Listing 9: Commands to generate an RSA keypair

Once the keys are generated, each pair of keys needs to be copied to the target IoT device. This is done using the Static file approach shown in Section 5.9, which will make both keys accessible as external variables in the application code. As the keys will become a part of the application code, they will be encrypted automatically if the flash encryption is enabled.

To make sure the application can locate all key files, they must be stored inside a data directory (which will be introduced in Section 8.7) and named `sta_priv.pem`, `sta_pub.pem`, `conf_priv.pem`, and `conf_priv.pem`. The files starting with `sta_` will be copied to the Enrollee, and the files starting with `conf_` will be copied to the Configurator. The files containing `pub` in their name shall contain the public keys, and the files containing `priv` in their name shall contain the private keys.

The last key file that is present in this directory is the `sta_required_conf.pem`. This file will be copied to the Enrollee and it is used to configure the mutual authentication. If the mutual authentication should be enabled, this file should contain the public key of the Configurator that will be required by the Enrollee during the provisioning process. If mutual authentication should be disabled, this file should be empty.

8.2 Network credentials storage

Just like in the previous section, this section will use the Static file approach to provide the Configurator with a list of accepted Enrollees and theirs network credentials. To save space on the Configurator device, each Enrollee will be represented by an SHA-256 hash of its public key. The SHA-256 hashes will be stored in the Configurator device in a binary format, as will be shown later.

To make the configuration user-friendly, the configuration is defined using a text file, which will be converted into a binary format once the text file is completed. An example of this text file is shown in Listing 10. It is recommended to name this file `allowed_stas.txt` and save it in the same data directory as in the previous section.

```
59fb5e8fe8282eeeec86112     my_ssid;user1;password1
59aa5e00e83327066c86111     my_ssid;user2;password2
b1a4c42ef19374c36cebbad     other_ssid;user3;password3
```

Listing 10: A configuration file used to set the list of accepted Enrollees and their network credentials (the SHA-256 hashes were shortened for simplicity)

The format of this file is simple – each of the lines represents a single station (Enrollee). Each Enrollee's entry contains an SHA-256 hash of the Enrollee's public key and its connection string. Both values must be separated by at least one space. The connection string contains the SSID name, username, and password needed to connect to the network. Semicolons (`;`) are used to separate the components of the connection string. It is expected that the SSID and username do not contain semicolons. In the password, semicolons may be used without escaping.

Once the configuration file is prepared, a bash script can be used to convert the file into binary format. If the recommended filename and location were used for the text file, the script can be executed without any parameters (`./convert_allowed.sh`), which will create a binary file `allowed_stas.bin` with the same data. This file is then copied to the Configurator with the application automatically. Just like in the previous section, this file will be encrypted if the flash encryption is enabled.

The format of the binary data for one Enrollee is shown in Table 8.1. For multiple Enrollees, the binary representations of individual Enrollees are concatenated without any separators.

Type	Field	Offset	Size	Description
uint8[]	pubkey_hash	0 B	32 B	Hash of the public key (in binary form)
uint16	conn_str_len	32 B	1 B	Length of the connection string (in bytes)
uint8[]	conn_str	33 B	any	The connection string, NULL-terminated

Table 8.1: Structure of the Public key reply message

The reason for including the length of the connection string in the structure when the connection string is NULL-terminated is to speed up the lookup of a given entry based on the value of `pubkey_hash`. The algorithm to find an entry in this list is implemented by iterating over the list sequentially, comparing the value of `pubkey_hash` with the target hash, and skipping the connection string based on the length stored in `conn_str_len` if the hashes did not match.

8.3 Cryptography wrapper

This section introduces the cryptography wrapper module implemented in files `crypt.c` and `crypt.h`. The purpose of this module is to provide an easy-to-use interface to commonly cryptographic operations in this project. The module is implemented by calling the appropriate functions of the mbedtls library that was introduced in Section 5.6.

The cryptographic module is initialized by calling the `crypt_init_rand_generator` function, which initializes the random number generator, which is needed to encrypt or decrypt messages. The initialized generator is stored internally in the cryptography wrapper and applied when needed.

To decrypt messages, the wrapper also needs to be initialized with the private key using the `crypt_init_private_key` function. This approach was chosen because a single IoT device uses only one private key to decrypt all incoming messages, thus it would be superfluous to require the private key every time a message is decrypted. For encryption of messages, though, the public key needs to passed every time a message is encrypted, because the public key depends on the receiver of the message, and thus changes over time. The functions designated for messages encryption and decryption are `crypt_encrypt` and `crypt_decrypt`. Apart from the public key (for encryption only), the message to be encrypted or decrypted, and an output buffer must be passed to these functions.

Next to the encryption functions, the wrapper also provides a function `crypt_random` to generate random numbers. This function only requires an output buffer and the expected length of the generated number in bytes.

The last responsibility of the cryptographic wrapper is to provide functions for the calculation of SHA-256 hashes. The wrapper provides two functions for this purpose, `crypt_hash` and `crypt_get_pubkey_hash`. Both functions require the message to be hashed, its length, and a 256-bit output buffer. The former function will take the input buffer as-is, calculate its SHA-256 hash, and store the result into the output buffer. The `crypt_get_pubkey_hash` function is designed to calculate a hash of a public key. The application stores public keys in a format that is compatible with the mbedtls library, which is the public key in a PEM format followed by a NULL byte. However, to get the same results of the public keys' hashes as from any commonly used utility (e.g. the command `sha256sum *.pem` on Linux), the last NULL byte must not be used in the hash calculation. Thus, the `crypt_get_pubkey_hash` function calculates the hash of the input buffer excluding its last byte.

8.4 Network stack

This section explains the challenges that had to be addressed when sending messages over the ESP-NOW protocol that was introduced in Section 5.5. This section will be divided into several subsections based on the individual challenges. All functions relevant to sending ESP-NOW messages are implemented in files `wifi.c` and `wifi.h`.

8.4.1 Wi-Fi initialization

It is necessary to initialize the Wi-Fi module before it can be used to transmit ESP-NOW messages (it is not needed to connect to an Access Point, the initialization of the Wi-Fi module on the chip is sufficient). This is an easy task to do, as the manufacturer provides an example application[1] demonstrating a basic communication over ESP-NOW.

However, a bug was found in this example, because the example does not set the channel of the Wi-Fi interface. As the ESP32 chip uses an auto-channel selection to find a non-busy channel, it may be possible for different devices to select different channels. If the devices do not use the same channel, the ESP-NOW messages will not reach the destination. This was fixed by setting the Wi-Fi channel on all devices manually and this bug was reported to the manufacturer.

8.4.2 Task for processing ESP-NOW messages

The ESP32's Wi-Fi interface uses callback functions to inform the application that a new packet was received or transmission of an outgoing packet was finished. These callback functions are, however, called from a high-priority Wi-Fi task and it is strongly discouraged to perform any lengthy operation from this task, so the Wi-FI interface is ready to transmit more packets.

Because of this, a task with a lower priority for message processing must be created. This low-priority task is named `wifi_task`. This task stores all requests to process incoming and outgoing messages in a `s_wifi_queue` queue. The role of this task will be shown in the following sections 8.4.3 and 8.4.4.

8.4.3 Sending a message

The `wifi.h` file provides an interface for sending unencrypted or encrypted unicast messages and unencrypted broadcast messages. This interface is called by a file that implements the Enrollee's or the Configurator's operation (`sta.c` or `configurator.c`). An example sequence diagram of the calls necessary to send a unicast unencrypted message is shown in Figure 8.1.

The ESP-NOW protocol requires that the Wi-Fi interface keeps track of its peers. This is achieved by the functions `add_peer(mac)` and `add_peer_broadcast()` that call the corresponding system function. The interface also provides the functions `del_peer(mac)` and `del_peer_broadcast()` to delete a previously added peer.

To send a message, either of the functions `send_unicast`, `send_unicast_encrypted`, or `send_multicast` needs to be called. All functions require the data to be sent and their length, the unicast functions also require the destination MAC address, and the encrypted unicast function also requires the public key to encrypt the message. All three

[1] https://github.com/espressif/esp-idf/blob/master/examples/wifi/espnow/

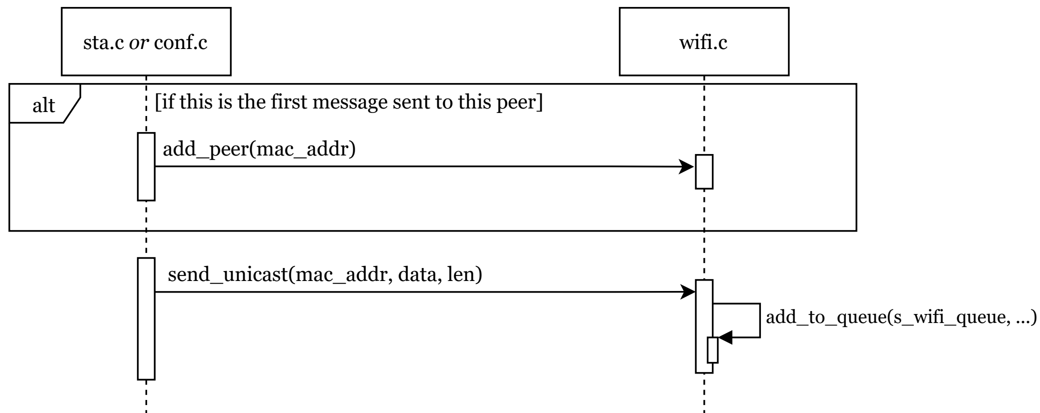


Figure 8.1: Function calls needed to send an unencrypted unicast message

functions will add an entry requesting the message to be sent to the `s_wifi_queue` (the `send_unicast_encrypted` function will encrypt the message first).

The `wifi_task` task will retrieve this request from the queue, and verify the length of the outgoing message. If the message is shorter than 250 bytes, the message will be sent using the system `esp_now_send` function. If the message is longer than 250 bytes, it will be fragmented using the approach defined in Section 7.4, and the fragments will be sent using the system `esp_now_send` function. Note that the fragmentation is not implemented for broadcast messages, because all broadcast messages sent by this protocol are guaranteed to be shorter than 250 bytes.

8.4.4 Receiving a message

The reception of messages is a bit more complex the sending of them because fragmented messages must be reassembled and it is necessary to work with callback functions. This section describes the actions that are taken from the moment an incoming message is registered by the network interface. Before this occurs, the operation code must register a callback function using the `set_incoming_message_callback` function from `wifi.h`.

Inside a high-priority system task

The `wifi.c` file registers its internal function as a callback from the network interface to receive all incoming ESP-NOW messages. As this callback is called from a high-priority system task that is not intended for time-consuming operations, this function will only create an entry about the incoming messages and adds it to the `s_wifi_queue`.

Inside the low-priority `wifi_task` task

The entry about the incoming message will be read from the `s_wifi_queue` by the user-defined low-priority `wifi_task` task. This task will use the first by of the message to identify whether this message is a part of a fragmented (multipart) message. If it is not, the message will be passed to the operation code. Otherwise, the `wifi.c` module will look up its internal list `s_incoming_multipart_msgs` for any previously received parts of this multipart message. If all parts of the multipart message are received, the assembled multipart message will be passed to the operation code and removed from the internal list of

incoming multipart messages. The operation code will not be able to recognize whether the received message was fragmented (except by comparing the length of the received message). If no new parts of the fragmented message are not received for one second, the internal entry about the incoming message will be deleted.

Inside the Enrollee's or Configurator's operation code

When the received message is passed into the Enrollee's or Configurator's operation code, the operation code will determine the type of the message based on its first 10 bytes and validate the length of the message based on the message's type. For some types of messages, the code may also call the function `decrypt_received_message(msg)` in `wifi.h`, which will decrypt the message in place. Then, the operation code will typically call a message-type-specific function to process the message.

8.5 STA (Enrollee) operation

This section explains the operations taken by the Enrollee-specific code. This code is implemented in the `sta.c` and `sta.h` files. The overall operation of this code is represented by the finite state machine (FSM) shown in Figure 8.2. The following sections will describe the actions taken in each state of the FSM.

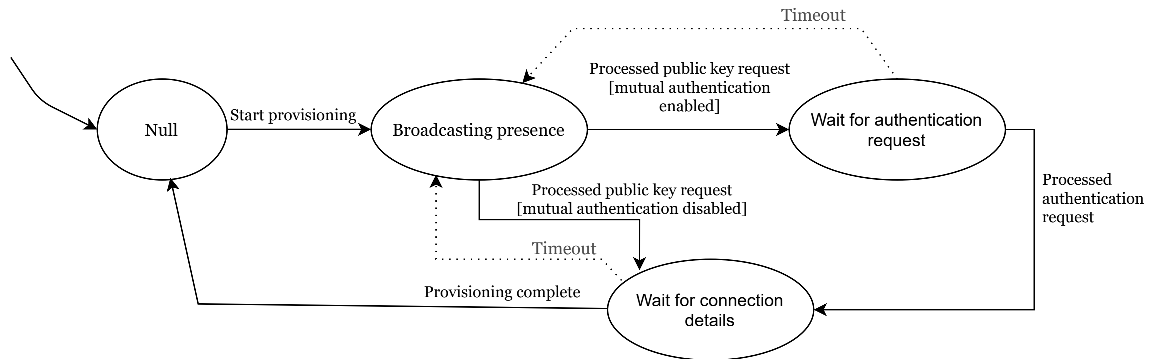


Figure 8.2: Enrollee's finite state machine

8.5.1 The initial (null) state

The initial (or null) state indicates that the provisioning process is not currently running, either because it was not initialized yet or it was already completed. In this state, the program will check in the non-volatile storage whether the provisioning process was already completed (this does not apply if the station is configured to rerun the provisioning process). If it was not, all necessary modules (namely the cryptography and network modules) will be initialized, the presence announcement message will be prepared, and the provisioning process will be started, changing the current state to *Broadcasting presence*.

8.5.2 Broadcasting presence

In the *broadcasting presence* state, the application will start a timer that will periodically broadcast the presence announcement message every 3 seconds, until the timer is stopped because the *Public key request* message from the Configurator was received.

When the *Public key request* is received, the station will reply with a *Public key reply* message that contains the public key of this station, and change its state to *Wait for authentication request* if the mutual authentication is enabled or to *Wait for connection details* otherwise.

The station will remember Configurator's MAC address in `s_configurator_mac` when it leaves the *broadcasting presence* state and ignore messages from other devices unless the status returns to the *broadcasting presence* state. The station will return to the *broadcasting presence* state if no message is received from the Configurator for 2 seconds.

8.5.3 Wait for authentication request

The station will reach the *Wait for authentication request* state if the mutual authentication is enabled and the station has processed the *Public key request* message. In this state, the station will wait for the *Authentication request* message from the Configurator (for up to 2 seconds).

If the authentication request is received, the station will decrypt the message, and reply with an encrypted *Authentication reply* message. This will reset the inactivity timer and change the station's state to *Wait for connection details*.

8.5.4 Wait for connection details

In this state, the station waits for the *Connection details* message. When it is received, the station will validate the correctness of the received nonce if the mutual authentication was enabled. The message will be ignored unless the received nonce matches the nonce sent in the *Authentication reply* message in the previous state.

Regardless of whether the mutual authentication was enabled, the station will validate the format of the received connection string, and the SSID, username, and password will be extracted from it. All of these values will be stored in the non-volatile storage, thus completing the provisioning process.

8.6 Configurator operation

The Configurator device never performs any actions on its own (apart from deleting inactive entries), because it only responds to request from Enrollees. However, as the provisioning protocol is stateful, the Configurator device must be able to track the current state of provisioning for each Enrollee. This is done by the `s_provisioned_stas` list which contains the status of the provisioning protocol for each Enrollee that has sent any message in the last five seconds. The structure of the entry for one Enrollee is shown in Table 8.2. All entries will be removed if the counterpart device is not active for 5 seconds. Note that some fields of the structure will not be filled in the early stages of provisioning.

The following sections will describe how the individual messages that can be received from Enrollees are handled on Configurator.

Type	Field	Description
`uint8[]`	`mac_addr`	Enrollee's MAC address
`TimerHandle_t`	`timer`	Timer to check for the Enrollee's inactivity
`enum`	`state`	State of the provisioning process (one of: `WAIT_FOR_PUBKEY`, `WAIT_FOR_AUTH_REPLY`, or `DONE`)
`uint8[]`	`pubkey_hash`	SHA-256 hash of the Enrollee's public key
`mbedtls_pk_context`	`pubkey`	Enrollee's public key
`bool`	`mutual_authentication`	Whether mutual authentication is enabled
`uint8[]`	`nonce_1`	The nonce sent in the Authentication Request message

Table 8.2: Structure of the entry to store the state of the provisioning protocol for one Enrollee

8.6.1 Processing the Presence announcement message

When a *Presence announcement* message is received, the Configurator will first check that it does not have an entry that contains the provisioning status of this device in the `s_provisioned_stas` list. If it does, it means the provisioning protocol is already running and the message will be ignored. Then, the device will check that all preconditions needed to start the provisioning protocol as defined in Subsection 7.5.2 are satisfied.

If the provisioning protocol may be started, the Configurator will initialize a new entry that will store the status of provisioning of this device. In this entry, the state will be set to `WAIT_FOR_PUBKEY`, the inactivity timer will be initialized, and the MAC address, hash of the public key, and the mutual authentication flag will be filled.

Finally, the device will respond with the *Public key request* message.

8.6.2 Processing the Public key reply message

When a *Public key reply* message is received, the Configurator will first check that the status entry for this Enrollee exists and its state is set to `WAIT_FOR_PUBKEY`. If it is, it will reset the inactivity timer and verify the received public key matches the hash that was sent in the *Presence announcement* message.

If the verification is successful, the Configurator will store the public key and change the state to `WAIT_FOR_AUTH_REPLY` if the mutual authentication is enabled or to `DONE` if it is disabled.

If the mutual authentication is enabled, the Configurator will generate a nonce that must be repeated by the Enrollee and send the *Authentication request* message. Otherwise, the device will send the *Configuration details* message. In both cases, the message will be encrypted by the Enrollee's public key.

8.6.3 Processing the Authentication reply message

When an *Authentication reply* message is received, the Configurator will first check that the status entry for this Enrollee exists and its state is set to `WAIT_FOR_AUTH_REPLY`. If it is, it will reset the inactivity timer and verify the received nonce matches the nonce that was sent in the *Authentication request* message.

If the verification is successful the Configurator will generate and send the *Configuration details* message and set the state to `DONE`. Note the destruction of the status entry will be done by the inactivity timer when it expires, so the *Configuration details* message may be retransmitted within this interval if needed.

8.7 Program configuration and execution

This section explains the steps needed to configure and build this application. First, the ESP32 development toolkit must be downloaded, installed, and initialized using the instructions in Section 5.2 and the target device must be set using the command shown in Listing 1 in Section 5.3.

Then, the RSA keys should be generated using the `generate_keys.sh` script found in the project's data directory, which is located in `project-dir/main/data/`. The generated keys will be saved in the same directory. The generated files do not need to be moved or renamed, as they will be compiled with the project.

In the same dir, the `sta_required_conf.pem` and `allowed_stas.bin` files should be also updated. The `sta_required_conf.pem` file configures the mutual authentication on the Enrollee. If the file is not empty, the Enrollee will only accept credentials from a Configurator with a given public key. To enable mutual authentication, the content of `conf_pub.pem` should be copied into this file. Otherwise, the `sta_required_conf.pem` file should be empty.

The `allowed_stas.bin` file can be updated by modifying the `allowed_stas.txt` file and running the `convert_allowed.sh` script without parameters. Details on the format used by the `allowed_stas.txt` were given in Section 8.2.

Next, the `idf.py menuconfig` command can be executed in the project's root folder (`project-dir/`) to configure the project. The relevant options are located in the **IoT provisioning configuration** section of the configuration menu. The following values can be configured using this tool:

- **Role of device** – sets whether a Station (Enrollee) or a Configurator device will be compiled
- **Wi-Fi Channel** – the Wi-Fi channel to use. All devices must use the same channel.
- **Force mutual authentication** – if enabled, the Configurator will only accept Enrollees with mutual authentication enabled. This value has no effect on Enrollees.
- **Force reconfiguration** – if enabled, the Enrollees will redo the provisioning process even if it is already configured. This value has no effect on Configurators.

Finally, it is recommended to enable the flash encryption and the secure boot on the Configurator device to prevent the network credentials from being leaked from the device if an attacker would gain physical access to the Configurator.

Once the project is configured, it may be built and flashed into the device using the command `idf.py -p /dev/ttyUSB0 flash monitor`, where `/dev/ttyUSB0` is the USB port the ESP32 device is connected to. See Section 5.3 for more information on building and flashing projects.

Chapter 9

Evaluation

This section evaluates the project's design and implementation. First, Section 9.1 reviews how the requirements from the Problem definition in Chapter 2 were addressed. Then, Section 9.2 discusses the user experience of the implemented solution, and Section 9.3 reviews the security aspects of the implementation. And finally, Section 9.4 discusses possible extensions of the implementation.

9.1 Fulfillment of requirements

This section lists the requirements for the implementation of the provisioning protocol that were defined at the end of Chapter 2, and reviews how each of the requirements was addressed.

- *There should be a dedicated configurator device that will store all necessary network credentials, and provide them to all legitimate IoT devices, rather than forcing the user to configure the credential details on each device separately.*
  This is indeed the key aspect of the implemented provisioning protocol, as a dedicated Configurator device is used to provision all Enrollees.

- *As most IoT devices have little to no user interface, the implementation should not require any unnecessary user action on the IoT device that is provisioned.*
  The provision protocol in question had to make the decision between security and usability. While it would be trivial to create a provisioning protocol that does not require any interaction with the IoT device apart from connecting it to the power supply – all that the Configurator would have to do is to broadcast the network credentials in plaintext – such a solution would not be secure. There must a mean for the Configurator to determine whether it is communicating with a legitimate Enrollee. To do this, each Enrollee needs to possess some kind of unique identifier that will allow the Configurator to identify the Enrollees. There will be more discussion on the usability of the solution given in Section 9.2.

- *The provisioning protocol should be secure. It must not be possible for an unauthorized IoT device to obtain the network credentials. Other security vulnerabilities should be mitigated as well.*
  It is not possible for an unauthorized IoT device to obtain the network credentials, because the protocol always sends the network credentials encrypted using the En-

rollee's public key which has been known the Configurator beforehand. More details on the security of the protocol will be discussed in Section 9.3.

- *The provisioning protocol must anticipate that the newly provisioned device has no internet connectivity, as the device did not receive the network credentials yet. Thus, an alternative way for the newly provisioned device and the configurator device to communicate must be found.*
  This is solved by using the ESP-NOW protocol, which is connectionless.

- *The IoT device should only engage in the provisioning protocol if it had not received the network credentials yet.*
  The Enrollee stores the received network credentials in non-volatile storage and does not repeat the provisioning process if previous credentials are saved. It is possible to override this behavior though.

9.2 User experience

While the provisioning protocol works without any user interaction at all, there are some prerequisites that must be satisfied that reduce the user experience. This section will first cover the prerequisites that must be satisfied on the Configurator device, and then the Enrollee's prerequisites will be covered.

On the Configurator device, it is first necessary to generate a pair of RSA keys that will be used if mutual authentication is required. This key-pair must be generated even if there is no intent to use the mutual authentication yet because the mutual authentication might be required by some Enrollee in the future. However, the generation of these keys is simple, as it requires running one bash script and the keypair only needs to be copied to one device.

The second object that needs to be configured on the Configurator device is a list of the Enrollees and their network credentials. The difficulty of the generation of this list depends on how the user currently stores the devices' network credentials. As the list is maintained in a form of a text file, it may be possible for the user to create a script that will auto-generate the content of this file from their source of credentials.

The configuration needed **on the Enrollee device** depends on whether the mutual authentication is enabled. In both cases, the Enrollee needs to generate a pair of RSA keys that will be used in the provisioning protocol. This key-pair may, however, be generated by the device manufacturer or other third party, and the user may receive a list of the public keys (or their SHA-256 hashes) of all devices. If mutual authentication is enabled, the Enrollees also must be provided with the Configurator's public key.

Table 9.1 displays the amount of time needed for the Enrollee device to reach individual stages of the provisioning protocol, either with the mutual authentication enabled (in column 2) or disabled (in column 3). In both cases, the device needed approximately 690 milliseconds to boot and initialize the Wi-Fi interface. Then, it took 30 ms to send the Presence announcement message, process it on the Configurator device, and send the public key request. After this, the needed 660 ms to send the Authentication request and another approximately 600 ms to send the Connection details message. This delay occurs because the IoT device needs to process lengthy public-key cryptography operations.

After this, other scenarios were tested as well, such as the case when the Enrollee device already had received the network credentials on the previous boot or when the Enrollee's

Step	With mutual authentication	Without mutual authentication
Device boot	437 ms	417 ms
Wi-Fi initialization	687 ms	687 ms
Received public key request	717 ms	717 ms
Received authentication request	1 377 ms	-
Device is fully provisioned	1 907 ms	1 387 ms

Table 9.1: Overall time reach individual steps of the provisioning protocol, either with or without mutual authentication enabled

public key did not match the Configurator's settings or vice versa, and more. The devices behaved in all tests as expected.

9.3 Security considerations

This section overviews the attacks possible on this implementation. The first two subsections will consider the possible attacks on the protocol itself depending on whether the mutual authentication is enabled. Then, Subsection 9.3.3 will consider the vulnerabilities when the private keys and network credentials are copied to the IoT device. And finally, subsections 9.3.4 and 9.3.5 describe the threats when an attacker gains physical access to the device.

9.3.1 Attacks on the protocol when mutual authentication is enabled

When mutual authentication is enabled, the protocol is resistant to impersonation and man-in-the-middle attacks, because both devices known each other's public keys beforehand, and public-key cryptography is used to encrypt sensitive data.

9.3.2 Attacks on the protocol when mutual authentication is disabled

When the mutual authentication is disabled, the Enrollee is not able to verify that it has received the connection details from a genuine Configurator. Depending on the scenario, though, this may be easily recognizable as the Enrollee will not connect to the target network and perform the required task. The connection details are still protected from disclosure because the details are encrypted using the Enrollee's public key and the Configurator knew the SHA-256 hash of this key beforehand, thus it is not possible to trick the Configurator to encrypt the data by an attacker-chosen public key.

9.3.3 Attacks on application deployment

Extra caution needs to be taken on the storage of private keys and network credentials before these details are copied into the IoT device. As both the private keys and network credentials are stored in unencrypted text files, it is necessary to delete these files once the devices are configured. It is also necessary to delete all build files using `idf.py fullclean`, as the configuration files are copied to the `build/` folder during the compilation.

9.3.4 Attacks on the Enrollee device

The implemented application does not encrypt the private key or network credentials on the ESP32 device by default to ensure any user application that will be executed when the provisioning process is complete can read the received network credentials. The target application should encrypt the network credentials and either encrypt or delete the private provisioning key when its first run. It is also possible to enable the flash encryption and secure boot on the device to make sure the private key is encrypted before the device is provisioned, but this may pose consequences for the target application.

9.3.5 Attacks on the Configurator device

On the Configurator device, it is strongly recommended to enable the flash encryption and secure boot to make sure the network credentials of all devices cannot be obtained by reading the content of the flash storage if an intruder gains physical access to the Configurator. If the flash encryption is enabled, all data in the flash storage including the private keys and network credentials are encrypted using the AES cypher. The encryption key is stored in the eFuse and cannot be read by software.

The application code can, however, access the unencrypted network credentials and the private keys. The access to the network credentials cannot be avoided, as the provisioning protocol needs to be able to read those credentials and send them to another (Enrollee) device. The private key could be better protected by making it inaccessible to the application code, however, the private key is of little value if an attacker is able to manipulate the application code because that would also give them the ability to read the network credentials.

The secure boot option ensures that the attacker is not able to reflash the device with their own malicious application. However, as a reflash of the device deletes the old content of the flash storage, the attacker should not be able to retrieve the network credentials by flashing the device with an application that reads all bytes from the flash storage. However, the manufacturer recommends turning both flash encryption and secure boot options on simultaneously.

9.4 Possible extensions

The implemented provisioning protocol is capable of providing the network credentials to unconfigured IoT devices. With this implementation in place, the protocol can be further expanded in many ways, as will be shown in this section.

One example of such an extension is to create a user application that would speed up the configuration of the Configurator. If new IoT devices would be manufactured with pre-generated provisioning keys, each IoT could contain a sticker with a QR code that holds the SHA-256 hash of this device's public key. The user application could then scan this QR code and automatically configure the Configurator to trust the Enrollee for a specified amount of time.

The provisioning protocol can currently configure the Enrollee with network credentials that are based on a combination of a username and a password, such as the credentials for Open Authentication, WPA2-PSK, and some WPA2-Enterprise authentication protocols. The provisioning protocol could support other Wi-Fi authentication protocols in the future, such as the ones that use client certificates.

The implementation could also change the application's runtime settings based on the specific use-case of each device. Currently, the user application that is executed when the device is provisioned with network credentials must be stored in the Enrollee. As an alternative, the implementation could completely overwrite the application code of an IoT device based on the device's use-case using an over-the-air update – this way, the application code would not have to be pre-installed in the device.

Chapter 10

Conclusion

The aim of this thesis was to review current approaches used to provision new IoT devices with network credentials and to design and implement a new provisioning protocol that will securely provision IoT devices with minimal overhead, especially with regards to the amount of user interaction necessary to provision a new device.

The review of the existing provisioning solutions has shown that current IoT devices often create an Access Point on their first boot to which to user needs to connect from another device and fill in the network credentials in an IoT device-generated web form. Other provisioning options include the use of a dedicated Gateway device that implements as many IoT devices' APIs as possible and provides a single configuration interface; or a smartphone that broadcasts the network credentials to all nearby devices with no regard to security.

The thesis then introduced the upcoming Wi-Fi Easy Connect standard that aims to provide a solution for this problem, but this standard is too complex and not adopted by current devices.

Thus, a simpler provisioning protocol was designed in this thesis. The protocol recognizes two types of devices, an Enrollee and a Configurator. A network must contain at least one Enrollee and exactly one Configurator. The Configurator device is responsible for the storage of the Enrollees' network credentials and it will provide these details to the Enrollees upon request. An Enrollee is an unconfigured device that uses the services of the Configurator to obtain the network credentials.

As the Enrollee device has no internet connectivity at the moment, a connectionless ESP-NOW protocol was used to communicate during the provisioning protocol.

The protocol uses public-key cryptography to ensure only the authorized Enrollees are able to retrieve the Wi-Fi credentials from the Configurator. Each device is identified by a unique pair of RSA keys, and the Configurator device keeps a whitelist of permitted Enrollees' public keys. The protocol offers optional mutual authentication, in which the Enrollee device will also verify the Configurator's identity.

The implemented provisioning protocol provides resistance against the network credentials disclosure, as the credentials are transmitted encrypted by a key the is only known to the Enrollee. However, special attention must be given to the storage of the credentials on the IoT device. It is possible to encrypt the credentials on the Configurator device by enabling flash encryption and secure boot. The same option is available on the Enrollee, however, the implementations for the user application must be considered first before enabling this option.

The implementation is open for further extension. For example, it is possible to extend the Enrollee code to perform an OTA update to download a use-case specific application code once the device is provisioned with network credentials. Or, a graphical interface could be created to smooth the process of the configuration of the Configurator device.

Bibliography

[1] Amazon Web Services, Inc.. *Software Timers*. 2020. Accessed 01.05.2021. Available at: https://www.freertos.org/RTOS-software-timer.html.

[2] Amazon Web Services, Inc.. *Tasks*. 2020. Accessed 01.05.2021. Available at: https://www.freertos.org/implementing-a-FreeRTOS-task.html.

[3] Amazon Web Services, Inc.. *XQueueCreate*. 2020. Accessed 01.05.2021. Available at: https://www.freertos.org/a00116.html.

[4] Apple Inc.. *Bonjour*. 2013. Accessed 19.12.2020. Available at: https://developer.apple.com/bonjour/.

[5] Bellavista, P. and Zanni, A. Feasibility of Fog Computing Deployment Based on Docker Containerization over RaspberryPi. In: *Proceedings of the 18th International Conference on Distributed Computing and Networking*. Association for Computing Machinery, 2017. ICDCN '17. DOI: 10.1145/3007748.3007777. ISBN 9781450348393.

[6] Corporation, M. *UPnP Device Architecture V1.0 Annex A – IP Version 6 Support*. 2002. Accessed 20.12.2020. Available at: http://upnp.org/specs/arch/UPnP-arch-AnnexAIPv6-v1.pdf.

[7] Dustdar, S., Nastić, S. and Šćekić, O. Provisioning Smart City Infrastructure. In: *Smart Cities: The Internet of Things, People and Systems*. Springer International Publishing, 2017, p. 27–46. DOI: 10.1007/978-3-319-60030-7_3. ISBN 978-3-319-60030-7.

[8] Espressif Systems. *SmartConfig*. 2016. Accessed 04.01.2021. Available at: https://docs.espressif.com/projects/esp-idf/en/v4.2.1/esp32/api-reference/network/esp_smartconfig.html.

[9] Espressif Systems. *Unified Provisioning*. 2016. Accessed 04.01.2021. Available at: https://docs.espressif.com/projects/esp-idf/en/v4.2.1/esp32/api-reference/provisioning/provisioning.html.

[10] Espressif Systems. *Embedding Binary Data*. 2020. Accessed 05.05.2021. Available at: https://docs.espressif.com/projects/esp-idf/en/v4.1/api-guides/build-system.html#embedding-binary-data.

[11] Espressif Systems. *ESP-NOW*. 2020. Accessed 07.01.2021. Available at: https://docs.espressif.com/projects/esp-idf/en/v4.2.1/esp32/api-reference/network/esp_now.html.

[12] Espressif Systems. *ESP-TLS*. 2020. Accessed 10.04.2021. Available at: https://docs.espressif.com/projects/esp-idf/en/v4.2.1/esp32/api-reference/protocols/esp_tls.html.

[13] Espressif Systems. *ESP32-S2 Family Datasheet*. 2020. Accessed 06.01.2021. Available at: https://www.espressif.com/sites/default/files/documentation/esp32-s2_datasheet_en.pdf.

[14] Espressif Systems. *ESP32-S2-Saola-1*. 2020. Accessed 06.01.2021. Available at: https://docs.espressif.com/projects/esp-idf/en/v4.2.1/esp32s2/hw-reference/esp32s2/user-guide-saola-1-v1.2.html.

[15] Espressif Systems. *ESP32-S2 Technical Reference Manual*. 2020. Accessed 06.02.2021. Available at: https://www.espressif.com/sites/default/files/documentation/esp32-s2_technical_reference_manual_en.pdf.

[16] Espressif Systems. *Flash Encryption*. 2020. Accessed 12.04.2021. Available at: https://docs.espressif.com/projects/esp-idf/en/v4.1/security/flash-encryption.html.

[17] Espressif Systems. *Get Started*. 2020. Accessed 07.01.2021. Available at: https://docs.espressif.com/projects/esp-idf/en/v4.2.1/esp32s2/get-started/index.html.

[18] Espressif Systems. *Non-volatile storage library*. 2020. Accessed 05.05.2021. Available at: https://docs.espressif.com/projects/esp-idf/en/v4.2.1/esp32/api-reference/storage/nvs_flash.html.

[19] Espressif Systems. *Project Configuration*. 2020. Accessed 09.03.2021. Available at: https://docs.espressif.com/projects/esp-idf/en/v4.2.1/esp32/api-reference/kconfig.html.

[20] Espressif Systems. *Secure Boot V2*. 2020. Accessed 15.04.2021. Available at: https://docs.espressif.com/projects/esp-idf/en/v4.1/security/secure-boot-v2.html.

[21] Goasduff, L. *Gartner Says 5.8 Billion Enterprise and Automotive IoT Endpoints Will Be in Use in 2020*. 2019. Accessed 10.12.2020. Available at: https://www.gartner.com/en/newsroom/press-releases/2019-08-29-gartner-says-5-8-billion-enterprise-and-automotive-io.

[22] IEEE Standards Association. IEEE Standard for Information technology–Telecommunications and information exchange between systems Local and metropolitan area networks–Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. *IEEE Std 802.11-2012 (Revision of IEEE Std 802.11-2007)*. 2012, p. 1–2793. DOI: 10.1109/IEEESTD.2012.6178212.

[23] Kim, J. E., Boulos, G., Yackovich, J., Barth, T., Beckel, C. et al. Seamless Integration of Heterogeneous Devices and Access Control in Smart Homes. In: *2012 Eighth International Conference on Intelligent Environments*. 2012, p. 206–213. DOI: 10.1109/IE.2012.57.

[24] Kliem, A. and Renner, T. Towards On-Demand Resource Provisioning for IoT Environments. In: Nguyen, N. T., Trawiński, B. and Kosala, R., ed. *Intelligent Information and Database Systems*. Springer International Publishing, 2015, p. 484–493. ISBN 978-3-319-15705-4.

[25] Kwon, D., Hodkiewicz, M. R., Fan, J., Shibutani, T. and Pecht, M. G. IoT-Based Prognostics and Systems Health Management for Industrial Applications. *IEEE Access*. 2016, vol. 4, p. 3659–3670. DOI: 10.1109/ACCESS.2016.2587754.

[26] Lethaby, N. *Run-Time Provisioning of Security Credentials for IoT Devices*. 2020. Accessed 10.10.2020. Available at: https://www.electronicdesign.com/technologies/iot/article/21126707/runtime-provisioning-of-security-credentials-for-iot-devices.

[27] Mohamed, S., Forshaw, M. and Thomas, N. Automatic Generation of Distributed Run-Time Infrastructure for Internet of Things. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 2017, p. 100–107. DOI: 10.1109/ICSAW.2017.51.

[28] Nastic, S., Truong, H.-L. and Dustdar, S. Sdg-pro: a programming framework for software-defined iot cloud gateways. *Journal of Internet Services and Applications*. Springer. 2015, vol. 6, no. 1, p. 21.

[29] Nixon, T. and al., A. R. at. *Web Services Dynamic Discovery (WS-Discovery)*. 2009. Accessed 20.12.2020. Available at: https://docs.oasis-open.org/ws-dd/discovery/1.1/os/wsdd-discovery-1.1-spec-os.html.

[30] Rushton, M. *CloudInit*. 2019. Accessed 03.01.2021. Available at: https://help.ubuntu.com/community/CloudInit/.

[31] Sundmaeker, H., Guillemin, P., Friess, P. and Woelfflé, S. Vision and challenges for realising the Internet of Things. In:. Cluster of European Research Projects on the Internet of Things, 2010, chap. Strategic Research Agenda, p. 39 – 82.

[32] Suppatvech, C., Godsell, J. and Day, S. The roles of internet of things technology in enabling servitized business models: A systematic literature review. *Industrial Marketing Management*. 2019, vol. 82, p. 70 – 86. DOI: https://doi.org/10.1016/j.indmarman.2019.02.016. ISSN 0019-8501.

[33] Wi-Fi Alliance. *Wi-Fi Easy Connect*. 2020. Accessed 08.01.2021. Available at: https://www.wi-fi.org/discover-wi-fi/wi-fi-easy-connect.