Universität Zürich[UZH]

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Hardening IoT Devices: An Analysis of the ESP32 Microcontroller

Michel Sabbatini
Zürich, Switzerland
Student ID: 21-704-564

Supervisor: Thomas Grübl, Daria Schumm
Date of Submission: September 1, 2024

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

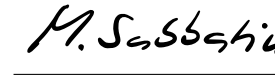
ifi

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Declaration of Independence

I hereby declare that I have composed this work independently and without the use of any aids other than those declared (including generative AI such as ChatGPT). I am aware that I take full responsibility for the scientific character of the submitted text myself, even if AI aids were used and declared (after written confirmation by the supervising professor). All passages taken verbatim or in sense from published or unpublished writings are identified as such. The work has not yet been submitted in the same or similar form or in excerpts as part of another examination.

Zürich, 01.09.2024



Signature of student

Kurzfassung

Die Zahl der Geräte im Internet der Dinge (IoT) ist in den letzten Jahren rasant gestiegen. IoT-Geräte sind häufig Low-End-Produkte, die von Hobbyanwendern genutzt werden und oft nicht ausreichend konfiguriert sind. Dadurch werden diese Geräte zu einem lukrativen Ziel für Angreifer, das bekannteste Beispiel dafür ist das Mirai Botnetz von 2016. Die ordnungsgemässe Konfiguration der Geräte, z.B. durch die Deaktivierung ungenutzter Protokolle und die Änderung von Standardkonfigurationen, ist eine einfache, aber sehr wirksame Massnahme zur Verbesserung der Gerätesicherheit. Bislang gibt es noch keine Lösung, die das Updaten der Sicherheitskonfiguration von low-end embedded Geräten wie dem ESP32-S3 automatisiert. Aktuelle Systeme zur Überwachung von IoT-Geräten konzentrieren sich in erster Linie auf die Analyse von Netzwerkpaketen. In dieser Arbeit wird eine Lösung vorgeschlagen, die die Sicherheitskonfiguration des ESP32-S3 Mikrocontrollers überwacht und sie bei Bedarf automatisch anpasst.

Die in dieser Arbeit implementierte Lösung heisst **ESP**32-S3 **D**evice **H**ardening **S**ystem, kurz **ESP-DHS**. Das ESP-DHS hat einen Server, der periodisch die Sicherheitskonfiguration eines ESP32-S3 Mikrocontrollers über das MQTT-Protokoll abfragt. Die Mikrocontroller-Firmware extrahiert die Konfiguration auf Anfrage und sendet sie an den Server zurück, wo sie analysiert wird. Der Server hat Zugriff auf den Quellcode der Firmware und aktiviert bei Bedarf Secure Boot, Flash Encryption und Memory Protection. Die aktualisierte Firmware wird dann kompiliert und auf den ESP32-S3 geflasht. Die gesamte Kommunikation zwischen dem Server und dem Mikrocontroller sowie alle vom Server durchgeführten Aktionen werden in einer Datenbank gespeichert. Das ESP-DHS bietet ausserdem Unterstützung für ein ATECC608B Secure Element, das kryptografische Funktionen und Möglichkeiten zum sicheren Speichern von kryptografischen Schlüsseln bietet.

Die ESP-DHS-Firmware nutzt 41% des 2 MB grossen Flash-Speichers des ESP32-S3. Die CPU ist im Ruhezustand 3% und während der Extraktion der Sicherheitskonfiguration 7% ausgelastet. Das Kompilieren und Flashen der Firmware dauert in der Testumgebung 1 Minute und 27 Sekunden. Um zu vermeiden, dass das ESP-DHS selbst zu einem Sicherheitsrisiko wird, ist es wichtig, den Zugriff auf den Mikrocontroller während des Flashens der Firmware zu beschränken und die Datenbank vor unbefugten Zugriffen zu schützen. In zukünftigen Arbeiten kann das ESP-DHS verbessert werden, indem mehr Mikrocontroller-Architekturen unterstützt und die Updates kabellos auf das Gerät geflasht werden.

Abstract

The number of Internet of Things (IoT) devices has been growing rapidly over the past few years. These devices are often low-end and deployed by amateur users that do not configure them properly. This makes them a lucrative target for attackers, the Mirai Botnet from 2016 being the most notable example of this. Configuring the devices properly, for example, by deactivating unused protocols and changing default configurations, is a simple yet very effective measure to improve device security. So far, a solution does not exist that automates updating the security configuration of low-end embedded devices like the ESP32-S3. Current IoT device monitoring systems focus primarily on the emitted network traffic. This thesis proposes a solution that monitors the security configuration of the ESP32-S3 microcontroller and adjusts it automatically if necessary.

The solution implemented in this thesis is called **ESP**32-S3 **D**evice **H**ardening **S**ystem, short **ESP-DHS**. The ESP-DHS has a server that periodically requests the security configuration of an ESP32-S3 microcontroller over the MQTT protocol. The firmware running on the microcontroller extracts the configuration upon request and sends it back to the server, where it is analyzed. The server has access to the source code of the firmware and activates secure boot, flash encryption, and memory protection if necessary. The updated firmware is then compiled and flashed onto the ESP32-S3. All communication between the server and the microcontroller and all actions taken by the server are logged to a database. The ESP-DHS also has built-in support for an ATECC608B secure element that provides cryptographic functions and secure key storage.

The ESP-DHS firmware uses 41% of the 2 MB flash memory of the ESP32-S3, 3% of CPU power in idle state, and 7% during configuration extraction. Compiling and flashing the firmware takes 1 minute and 27 seconds in the test environment. To avoid the ESP-DHS itself becoming a security risk, it is important to restrict access to the microcontroller during firmware flashing and to only grant database access to authorized entities. Future work could focus on improving the ESP-DHS by supporting multiple types of microcontrollers and flashing updates over the air.

Contents

Chapter 1

Introduction

1.1 Motivation

Security is an important requirement in any computer system but it is especially critical in IoT devices. Compromised IoT devices are not only a problem for the security and privacy of end users, but could also physically harm them when devices have sensors and actuators in the real world. Since the IoT relies heavily on interconnection between multiple devices, a compromised one is usually not isolated and can negatively impact the whole network [1]. An empirical study with over 1.3 million IoT devices found that more than 28 percent of them suffered from at least one known security vulnerability. Many IoT devices communicate frequently with MQTT servers, of which 88 percent were found not to be password protected due to misconfiguration [2].

Research showed that improper configuration is one of the main concerns in IoT devices. This includes deactivated security features like secure boot or flash encryption, activated but unused protocols or outdated protocols (e.g. HTTP instead of HTTPS or FTP instead of FTPS). A cause of this is are inexperienced users deploying devices without changing the default factory configuration [3]–[5]. By only adjusting the device configurations, [5] showed that three out of four devices could be protected effectively from the Mirai malware. Security standards like the NIST IR 8259A [6] or NIST SP 800-53 [7] also highlight the importance of proper device configuration.

Although device configuration is one of the simplest yet most effective device hardening techniques, current research about IoT device hardening does not reflect this importance. There exist a number of systems that monitor the network traffic or internal side channel information of IoT devices and that can isolate a device if necessary [8]–[10]. However, none of these systems take the configuration of the monitored devices into account. Prior work provides concrete steps for securing the device configuration but does not automate this process [3]–[5]. This leaves a gap for systems that monitor and automatically update the security configuration of IoT devices.

1.2 Description of Work

This work proposes the **ESP**32-S3 **D**evice **H**ardening **S**ystem, short **ESP-DHS**. The system automatically extracts, analyzes, and updates the security configuration of an ESP32-S3 microcontroller. The configuration is extracted by a part of the firmware running on the microcontroller called the ESP-DHS client and analyzed remotely by the ESP-DHS server. If the analysis reveals deactivated security features, the server activates them in the source code of the firmware before building and flashing the hardened firmware onto the microcontroller. All communication between server and client and all actions taken by the server are logged and stored in a database, which allows analyzing the events that happened in the system retrospectively. The ESP-DHS is a device monitoring tool that does not primarily focus on anomaly detection and isolation but takes a proactive approach to IoT device security. Meaning, that it takes action (i.e. updates the microcontroller firmware) before an anomaly is detected.

The implemented solution consists of five main parts: the server, the database, the firmware running on the microcontroller, the MQTT message broker, and the ATECC608B secure element. The firmware extracts the device security configuration and sends it back to the server upon request. The part of the firmware not responsible for the ESP-DHS logic is called the user application. The secure element is added to the ESP32-S3 to improve support for cryptographic functions and key storage, an example user application is implemented to show how these capabilities can be leveraged.

Since standards play an important role in the security of IoT systems, this work presents the most relevant IoT security standards. First, it is analyzed how the ESP32-S3 implements the capabilities from the NIST IR 8259A standard, then, it is elaborated on how selected NIST SP 800-53 controls are implemented in the ESP-DHS. This provides guidance for when the system is implemented in contexts where compliance with these security standards is essential, for example in large organizations.

As part of the evaluation, resource consumption, performance, and security enhancements are tested. The ESP-DHS firmware running on the ESP32-S3 uses 40.9% of the 2 MB flash memory and below 25% of the 0.7 MB of RAM. It only uses 3% of CPU power in idle state and up to 7% during the configuration extraction. On the test machine (MacBook Air M2 2022), activating security features and building and flashing the firmware takes 1 minute 27 seconds.

This work also discusses the potential attack surface the ESP-DHS introduces to the system. Attack vectors are the physical connection between the device running the ESP-DHS server and the ESP32-S3 as well as the unencrypted database. It is therefore important to deploy the ESP-DHS in an environment where physical access to the microcontroller is only possible by authorized parties and that the machine running the ESP-DHS server and database is protected.

Immediate future work should focus on improving the security of the ESP-DHS by using a database with access restrictions and flashing firmware over the air instead of using a physical UART connection. The ESP-DHS could be extended to support more types of microcontrollers and additionally monitor network traffic to detect malicious behavior.

1.3 Thesis Outline

This thesis is organized as follows: The background is provided in Chapter 2, which includes information about the devices, tools, and protocols used for the development of the ESP-DHS. Furthermore, the most relevant IoT security standards are introduced. In Chapter 3, current research about device hardening, secure elements, and IoT device monitoring and recovery is presented and discussed. Chapter 4 describes the design and functionality of the ESP-DHS as well as how IoT security standards are implemented in the system. Detailed information about all system components and their implementation are provided in Chapter 5. Resource consumption, performance, and effectiveness of the implemented solution are discussed in Chapter 6. In Chapter 7, the thesis is summarized, limitations of the ESP-DHS are discussed and potential future work is explored.

Chapter 2

Background

In this chapter, the necessary backgrounds for understanding this thesis are provided. First, selected aspects of the ESP32-S3 and its security features are presented and the technologies used for the implementation are explained. Then, an overview of the most relevant IoT security standards is given.

2.1 The ESP32-S3

ESP32 is a family of integrated, low cost and energy-efficient Systems on a Chip (SoC) by the Chinese company Espressif, first presented in 2016 [11]. These SoCs support Wi-Fi and Bluetooth as well as a wide range of peripherals such as multiple programmable GPIO pins and UART [12], making them a popular choice for various IoT applications [13].

Compared to other microcontrollers in the ESP32 lineup, the ESP32-S3 has AI acceleration support that helps with neural network computing, which makes it especially suitable for the Artificial Intelligence of Things (AIoT) market [14]. It has built-in security hardware, 700 KB of internal SRAM, and is powered by a dual Xtensa LX7 microprocessor, which can run at 240 MHz [12].

There exist multiple different versions of the ESP32-S3, in this work, the ESP32-S3-DevKitC-1 is used. Figure 2.1 shows a picture of the device, Table 2.1 describes its most important hardware components.

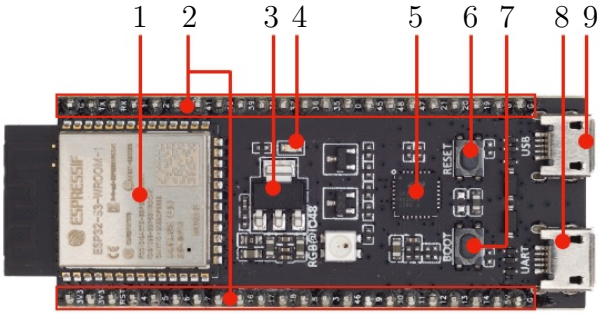


Figure 2.1: Important Hardware Components of the ESP32-S3-DevKitC-1

Label	Name	Description
1	MCU Module	Low energy ESP32-S3-WROOM-1 module with support for Wi-Fi, Bluetooth and a wide range of peripherals.
2	Pins	GPIO, power supply and ground pins to which multiple peripherals can be connected.
3	Power Regulator	Converts 5V power supply into 3.3V output.
4	Power On LED	Lights up when board has power through the USB connector.
5	USB-to-UART Bridge	Converts the received USB signal to an UART signal.
6	Reset Button	Causes the system to restart when pressed.
7	Boot Button	Pressing the reset button while holding the boot button will cause the device to download new firmware through the serial port.
8	USB-to-UART Port	Micro-USB port for power supply and communication with the chip by first going through the USB-to-UART bridge.
9	USB Port	Micro-USB port (USB 1.1 specification), provides power supply and direct communication with the chip, can be used for JTAG debugging.

Table 2.1: Description of the labeled Hardware Components of Figure 2.1 [15]

2.2 ESP32-S3 Security Features

The ESP32-S3 provides a variety of different security features [16], that are described in the following paragraphs. These features help to prevent untrustworthy code from being executed, secure the device's identity, provide secure storage for data, and ensure encrypted communication between devices.

2.2.1 eFuses

If activated, many of the security features cannot be deactivated anymore. This is due to eFuses, special bits in the ESP32 memory that are initially all 0 and cannot be reset anymore once set to 1. For debugging purposes, eFuses can be simulated in flash memory. eFuses can only be accessed by hardware and not by the software running on the CPU, making them a safe place to store confidential information, for example, the flash encryption key [17]. However, due to their one-time programmability, eFuses are not always a suitable solution and secure elements might needed, which is explained in more detail in Section 2.3.

2.2.2 Secure Boot

Secure Boot ensures that no untrustworthy code can be executed on the device by checking that every piece of software booted was signed with the private secure boot signing key. A digest of the public key is stored in eFuses. Signed application images have a signature block appended, which contains a signature of the preceding image and the secure boot public key. The first stage bootloader resides in ROM and therefore does not need to be checked, however, the second stage bootloader and application image both reside in flash and need to be signed. The first stage bootloader loads the second stage bootloader into RAM and verifies its signature, i.e. the public key digest in the eFuses is compared with the public key from the signature block and then it is checked whether the image signature is correct. If the verification is successful, the second-stage bootloader loads the application image and goes through the same verification procedure. Only if this check is also successful, the application will be started. If verification fails at one step and aggressive key revocation is activated, the public key digest in the eFuses will be marked as revoked and the corresponding private key cannot be used anymore to sign app images and bootloaders. This provides a high level of security against physical attacks on the device. Once activated, secure boot and aggressive key revocation cannot be deactivated since the respective eFuses are burned [18].

2.2.3 Flash Encryption

Flash Encryption encrypts the flash memory of the ESP32-S3 and therefore helps ensure the confidentiality of the stored firmware and data. When this feature is first activated, the firmware is flashed in plaintext and gets encrypted on the first boot using XTS-AES-256. This algorithm was specifically designed for disc encryption and addresses the weaknesses of e.g. AES-CTR. The flash encryption keys are stored in eFuses. Physically reading out the contents of the flash memory does not reveal its contents. When flash encryption is activated, some partitions are encrypted by default, including the second-stage bootloader, partition table, and application image. All other partitions need to be marked manually with an encryption flag. Flash encryption cannot be turned off once activated. For the best level of security, it is recommended to use flash encryption and secure boot in conjunction [19].

2.2.4 Memory Protection

Memory Protection is a low-level scheme that provides the ability to monitor memory access and raise an exception if any attempt that breaks these permissions is made. This feature is useful for preventing remote code injections [16].

2.2.5 Transport Layer Security

Transport Layer Security (TLS) support is built into the ESP32-S3, which should be used for all external communications. Taking advantage of this security feature does not only require activating the secure version of the desired protocol (e.g. HTTPS or MQTTS) in the device configuration but also changing the application code accordingly [16].

2.2.6 Other Security Features

The ESP32-S3 provides capabilities for the secure provisioning of devices in Wi-Fi systems, secure over-the-air (OTA) updates, and non-volatile storage (NVS) encryption. Furthermore, digital signatures based on RSA are supported with hardware acceleration. This helps establish a secure device identity to a remote endpoint [16]. These features are not explained in more detail since they are not an important part of the solution implemented in this thesis.

2.3 Secure Elements (ATECC608B)

Storing cryptographic keys in a secure environment is an important part of IoT security. The eFuses on the ESP32 are hardware protected and do not allow access by software running on the device. However, because of their one-time programmability, they are not suited for all kinds of applications [17]. To be more flexible while still adhering to good security practices, secure elements can be used. Secure elements have the capability to perform cryptographic operations like true random number generation or encryption/decryption tasks, and store keys with hardware protection. Applications use an interface to the secure element to feed the data on which a cryptographic operation shall be performed to the secure element, which then returns the result. This way, cryptographic functionality is physically separated from the rest of the application [20]. The secure element used in this thesis is the ATECC608B, a low-cost device by Microchip that provides the aforementioned functionality [21].

2.4 MQTT Protocol

Message Queuing Telemetry Transport (MQTT) is a lightweight publish-subscribe protocol, often used in the context of IoT applications. It is designed to have low latency in unreliable networks with low bandwidth. The broker is the central unit of any MQTT system and distributes the messages in the network. Clients can publish and subscribe to topics [22].

Figure 2.2 shows an example of how this works. A straight line denotes that a message was published, a dotted line denotes that a message was received. Client 1 and Client 2 are both subscribed to `/topic-1`. Client 1 is additionally subscribed to `/topic-2`.

When Client 1 publishes a message to `/topic-1`, the broker distributes the message to all clients that are subscribed to the topic, in this case, the message is returned to Client 1 and forwarded to Client 2. When Client 2 publishes a message to `/topic-2`, the message is only passed to Client 1 since it is the only client subscribed to this topic.

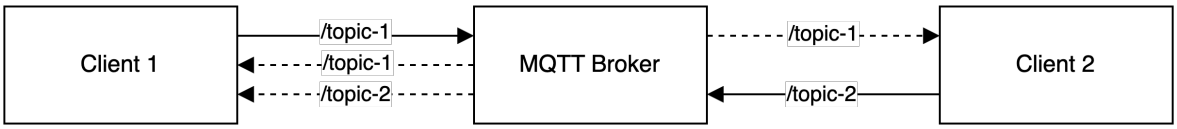


Figure 2.2: MQTT Protocol Example

2.5 ESP32 Application Development

2.5.1 ESP-IDF

Applications for the ESP32-S3 are developed using the Espressif Integrated Development Framework (ESP-IDF) [12]. The framework facilitates firmware development by providing capabilities for building, flashing, or configuring the project. It also helps with setting up a project by providing code snippets for different use cases or editing the eFuse table. The ESP-IDF Visual Studio Code extension offers many of these capabilities through a graphical user interface (GUI).

2.5.2 Project Configuration

ESP-IDF projects are configured using the Kconfig language [23]. Configuration options provide for example the Wi-Fi password or define whether secure boot is activated or not. Since it is a compile-time configuration system, firmware needs to be recompiled after configuration changes are made and the values are available in the code as preprocessor macros. Kconfig provides functionality for defining dependencies between options, default values, or grouping options. ESP-IDF provides a package that helps to edit and create such configurations easily [24].

Listing 2.1 shows an example of a Kconfig file that specifies a submenu in an ESP-IDF project. The `menu` keyword is followed by the title of the submenu. `ESP_WIFI_SSID` is the name of the configuration, its value is made available in the ESP-IDF project as a macro with the name `CONFIG_ESP_WIFI_SSID`. If no value is set manually, the default value “myssid” is used. The `help` keyword is followed by a short explanation of what the configuration is used for.

```
# Kconfig.projbuild

menu "DHS WiFi Configuration"
  config ESP_WIFI_SSID
    string "WiFi SSID"
    default "myssid"
    help
      SSID (network name) to which the program connects.
```

Listing 2.1: Kconfig Menu Example

In ESP-IDF projects, Kconfig files define the structure and options of the project configuration, sdkconfig files store the assigned values. Listing 2.2 shows an example of a sdkconfig file. Configuration can be edited using a graphical user interface or a command line tool provided by the ESP-IDF. When editing configuration with Python code, the Kconfiglib package can be used [25].

```
# sdkconfig

CONFIG_SOC_MPU_REGIONS_MAX_NUM=8
CONFIG_SOC_ADC_SUPPORTED=y
CONFIG_ESP_WIFI_SSID="myssid"
```

Listing 2.2: Sdkconfig Example

2.6 IoT Security Standards

In this section, the most important security standards in the context of hardening an ESP32 microcontroller are discussed, with an overview given in Table 2.2. This work focuses mainly on the NIST standards because they are freely available, but a comparison of how NIST SP 800-53 relates to ISO/IEC 27001 is given at the end of this chapter. One of the biggest challenges in IoT security is the fragmentation of security standards and regulations [26]. Many of the conventional security standards do not explicitly address the needs of IoT systems, but they can be adapted to do so [27].

2.6.1 Platform Security Architecture

First introduced in 2017 by ARM, the Platform Security Architecture (PSA) is a security framework that aims at bringing security best practices to IoT devices. PSA security certificates are issued by PSA Certified, a partnership between ARM and multiple other companies. Multiple device manufacturers, including Espressif, Mediatek, AWS, and Texas Instruments, have adopted PSA certificates for some of their devices. Complying with the PSA standards can give them easier access to global markets because they automatically comply with industry and government standards as well as IoT legislation. The certificates are not designed for a specific implementation or architecture, therefore they are applicable across a wide range of software, devices, and chips [28].

Standard	Description
PSA [28]	Provides IoT security best practices and guidelines for device manufacturers, silicon vendors, and system software providers. PSA Certificate Level 1 automatically ensures alignment with NIST IR 8259A and other standards.
NIST IR 8259A [6]	Provides device manufacturers with a basic set of device capabilities that are needed to protect IoT devices and systems.
NIST SP 800-53 [7]	Provides a rich set of privacy and security controls for organizations and information systems to protect them against human errors, hostile attacks, or structural failures amongst others.
ISO/IEC 27001 [29]	Provides a framework for implementing and maintaining an Information Security Management System (ISMS) within an organization and defines requirements the system must meet.

Table 2.2: IoT Security Standards Overview

An important concept for PSA is the Root of Trust (RoT). It describes a collection of inherently trusted functions that the remainder of the system or device can utilize to ensure security [28].

There exist four different levels for PSA certificates [28]:

- **Certification Level 1**
  This certification is for device, software, and chip vendors that want to show that sound security principles were applied. This certification can be achieved relatively quickly by choosing pre-certified silicon and software. The certification process requires the vendor to fill out a questionnaire, after which the device is reviewed in a PSA Certified laboratory. The ESP32-S3 is level 1 PSA certified [30] and therefore automatically aligns with NIST 8259A.

- **Certification Level 2 / 2+**
  With a level 2 certificate, chip vendors can show that their PSA Root of Trust (PSA-RoT) effectively safeguards against software attacks. Level 2+ additionally recognizes that cryptographic keys and operations are physically secured with a secure element.

- **Certification Level 3 / 3+**
  A level 3 certificate shows that a device is capable of defending against significant hardware and software attacks. A 3+ certification ensures a device protects cryptographic keys and operations physically.

- **Certification Level 4**
  A level 4 certified device has substantial protection against physical and software attacks due to an integrated secure enclave or an external secure element.

2.6.2 NIST IR 8259A

The NIST Interagency/Internal Report 8259A (NIST IR 8259A), also known as the "IoT Device Cybersecurity Capability Core Baseline", is aimed at device manufacturers and provides them with a set of basic security features that make their devices more secure if implemented. The document does not describe, how these capabilities must be implemented, leaving a lot of flexibility for the manufacturers [6]. Table 2.3 lists the six capabilities with a short description. Table 4.1 shows, how these capabilities are implemented in the ESP32-S3 Microcontroller.

It must be noted that just using a device that implements the measures provided by this report is not sufficient for achieving high security in a system. For example, the ESP32-S3 implements secure boot and flash encryption for data protection, but these security mechanisms are not enabled by default. In order to achieve high security, the developer needs a deep understanding of the device to configure it properly.

Capability	**Description**
Device Identification	The device can be uniquely identified, both physically and logically.
Device Configuration	The device's software configuration can be changed, but only by authorized entities.
Data Protection	Stored or transmitted data can be protected from unauthorized access and modification.
Logical Access to Interfaces	Logical access to the device's local and network interfaces is only granted to authorized entities.
Software Update	Authorized entities can update the device's software by using a secure configurable mechanism.
Cybersecurity State Awareness	The device can keep track of its security state and report it to authorized entities.

Table 2.3: NIST IR 8259A Capabilities [6]

2.6.3 NIST SP 800-53

The NIST Special Publication 800-53 (NIST SP 800-53), titled "Security and Privacy Controls for Information Systems and Organizations", is one of the most widely adopted information security frameworks in the world [31]. The current version is Revision 5, published in 2020. It provides a comprehensive set of security and privacy controls for organizations and information systems to protect them against a wide variety of threads, including human errors, hostile attacks or structural failures [7].

Table 2.4 gives an overview of the different categories of controls from NIST SP 800-53, Table 2.5 contains an excerpt of controls with a short description that are especially relevant in the context of hardening an ESP32-S3 microcontroller.

ID	Family
AC	Access Control
AT	Awareness and Training
AU	Audit and Accountability
CA	Assessment, Authorization, and Monitoring
CM	Configuration Management
IA	Identification and Authentication
IR	Incident Response
MP	Media Protection
PE	Physical and Environmental Protection
RA	Risk Assessment
SA	System and Services Acquisitions
SC	System and Communications Protection
SI	System and Information Integrity
SR	Supply Chain Risk Management

Table 2.4: Excerpt of NIST SP 800-53 Security and Privacy Control Families [7].

ID	Title	Description
CM-2	Baseline Configuration	Define and continually develop a baseline configuration of the system to ensure good security settings are employed.
CM-3	Configuration Change Control	Review proposed configuration changes and document them
SC-3	Security Function Isolation	Security functions must be isolated from non-security functions.
SC-8	Transmission Confidentiality and Integrity	Protect the confidentiality and integrity of transmitted information.
SC-13	Cryptographic Protection	Use cryptographic functions for digital signatures, random number or hash generation, or protecting sensitive information.
SI-4	System Monitoring	Monitoring the system to detect unauthorized connections or malicious application behavior.
SI-7	Software, Firmware, and Information Integrity	Ensure software and firmware integrity and provide security protection mechanisms against unauthorized code execution.
IA-3	Device Identification and Authentication	Devices should be authenticated and uniquely identified before establishing a network connection.

Table 2.5: Excerpt of relevant NIST SP 800-53 Controls [7]

2.6.4 ISO/IEC 27001

ISO/IEC 27001 is a standard by the International Organization of Standardization (ISO) and the International Electrotechnical Commission (IEC). ISO standards are internationally recognized and organizations can get certified against ISO/IEC 27001 [29].

Compared to NIST SP 800-563, ISO/IEC 27001 does not mainly focus on the implementation of specific security controls but rather provides organizations with a framework for implementing and maintaining an Information Security Management System (ISMS) [29]. An ISMS is a system to manage risks, protect data, and maintain cybersecurity compliance [32]. Annex A of ISO/IEC 27001 provides an overview of security controls that are necessary to implement this standard, ISO/IEC 27002 elaborates on these controls and provides more detail regarding their implementation [31]. According to [31], the controls of NIST SP 800-53 comprise a superset of ISO/IEC 27001. Table 2.6 gives a mapping between relevant NIST SP 800-53 controls and their ISO/IEC 27001 counterparts.

NIST SP 800-53 Controls	**ISO/IEC 27001 Controls**
CM-2	None
CM-7	A.12.5.1
SC-3	None
SC-7	A.13.1.1, A.13.1.3, A.13.2.1, A.13.2.1, A.14.1.3
SC-8	A.8.2.3, A.13.1.1, A.13.2.1, A.13.2.3, A.14.1.2, A.14.1.3
SC-13	A.10.1.1, A.14.1.2, A.14.1.3, A.18.1.5
SC-41	None
SI-4	None
SI-7	None
IA-3	None
IR-4	A.16.1.4, A.16.1.5, A.16.1.6
IR-5	None
SR-5	A.15.1.3

Table 2.6: Mapping of NIST Controls to ISO Controls [31].

Chapter 3

Related Work

In this chapter, solutions for securing IoT devices from recent scientific literature are discussed. Section 3.1 focuses on a variety of different approaches to IoT security like device hardening, security management servers, and frameworks for the IoT device hardening process. The two research works discussed in Section 3.2 are about device hardening solutions that address specifically the issue of secure key storage and provisioning in IoT devices. Section 3.3 discusses four different solutions for device monitoring and recovery.

3.1 Device Hardening Solutions

Various device hardening solutions are summarized and compared in Table 3.1. The following paragraphs describe the different solutions in more detail.

Research Work	**Solution Name**	**Devices**	**Summary**
Carillo et al. [3] (2023)	HALE-IoT	Higher-end devices e.g. Linksys EA6300v1	A framework for hardening legacy IoT devices by retrofitting defense systems into device firmware
Echeverría et al. [4] (2021)	–	Arduino Mega 2560, Raspberry Pi	A seven-step model to minimize the attack surface by executing device hardening processes
Kelly et al. [5] (2020)	–	IP Camera, Raspberry Pi, Siricam, VM Device	The Mirai malware was analyzed and four IoT devices were hardened against it by proper device configuration
Yoon & Kim [33] (2017)	–	–	An architecture for a remote security management server for IoT devices

Table 3.1: Device Hardening Solutions

HALE-IoT [3] is a framework that can retrofit security firmware on higher-end legacy IoT devices that do not receive manufacturer updates anymore. The HALE-IoT firmware is implanted into the file system of the device without having access to the original source code. HALE-IoT wraps the vulnerable services of the device with their more secure counterparts, e.g. HTTP is wrapped with HTTPS, FTP with SFTP, and Telnet with SSH. In between the wrapped services and the new secure front end, firewalls are installed for proactive attack detection. Furthermore, a secure admin interface for device management is provided. To evaluate its effectiveness, almost 400 emulated firmware images and four physical devices got HALE-IoT implanted, showing that the devices continued working as expected while also having high detection and prevention rates of attacks.

To approach the device hardening process in a structured manner, the authors of [4] propose a seven-step model. The seven steps are:

1. Defining the purpose and requirements of the device
2. Assessing the device risks
3. Disabling unnecessary protocols, services, and configuration
4. Determining the device's attack surface
5. Finding vulnerabilities
6. Conducting the device hardening
7. Evaluation

The risks, attack surface, and vulnerabilities can be quantified in order to receive an overall maturity level of the device on a scale from 0 (bad) to 4 (good). The model was tested in two case studies, the applied hardening methods include remapping port numbers, disabling unused file systems, configuring software updates, activating secure boot, configuring a firewall, and blocking the network interface when an unknown device was connected. It could be shown that with these measures, the device is not susceptible to a wide range of cyberattacks anymore.

In [5], hardening techniques against the Mirai botnet were proposed, implemented, and tested. First, a secure test environment consisting of four IoT devices was set up, then, the malware was deployed on these devices, which were all running on their default configurations. The authors show that three out of the four devices were vulnerable to the malware and got infected. By analyzing the infected devices, the behavior of the malware and the used attack vectors could be determined. Based on this analysis, the device was hardened by adjusting its configuration. Several hardening methods were suggested: Changing the default credentials, using SSH instead of Telnet, changing the Telnet port if the protocol cannot be deactivated, using HTTPS instead of HTTP, blocking access to certain IP addresses/ports that were used by the malware and disable the SMTP and FTP protocols. These countermeasures proved to be effective as none of the four devices could be infected anymore after hardening. This shows that proper device configuration can have a great impact on the security of an IoT device.

The authors of [33] propose an architecture for a remote security management server that manages various security functions of IoT devices remotely in order to make them more secure. The server consists of 16 different modules, for example, a Privacy Protection Module, Secure Storage Module or Crypto Engine Module. Four modules are especially relevant for this thesis: The Firmware Update Module provides functionality to keep the firmware on the IoT device up-to-date. The Security Policy Management Module analyzes the device's resources and sets a policy for the device, including an appropriate security configuration. The Vulnerability Assessment Module checks the device's configuration and searches for common and new vulnerabilities. The Device State Monitoring Module continuously checks the state in which the device currently is and uses this information to detect intrusions by hackers. No actual implementation of the architecture or some of its modules was provided.

The papers from Table 3.1 have different approaches to IoT device hardening. Except for [33], concrete implementations were provided and tested. Both [5] and [4] suggest measures that can be taken to make IoT devices more secure. In many IoT devices, these measures can be implemented relatively easily. However, for legacy IoT devices that do not receive firmware updates anymore and whose firmware code is not accessible, it can be difficult to implement them. Therefore, [3] provides a framework for the implementation of these measures by retrofitting security firmware on higher-end IoT devices. The identification of security vulnerabilities and the implementation of the device hardening measures were performed manually in all these papers, which may lead to bad scalability of these approaches.

3.2 Secure Key Generation and Storage

There exist two main approaches for the problem of secure key generation and storage on IoT devices: offloading the key generation to other devices or using a crypto coprocessor. Table 3.2 summarizes and compares these two solutions.

Research Work	**Solution Name**	**Devices**	**Architecture**	**Advantages/Drawbacks**
McPherson & Irvine [34] (2020)	–	MSP430G2553 with Smartphones	Offload key generation and provisioning to smartphones	+ No additional hardware required + Deployable through firmware update
Pearson et al. [20] (2020)	SIC^2	ESP32 with ATECC608A	Cryptographic operations and key storage on crypto coprocessor	+ lower energy consumption and faster TLS handshake – Additional hardware required

Table 3.2: Key Generation and Storage Solutions

Since low-end IoT devices usually cannot create suitable entropy to generate secure keys, [34] propose an architecture that leverages the capabilities of smartphones in the same

network for this purpose. The architecture can be deployed to any low-cost IoT device by a firmware update because no additional hardware is required. The architecture was tested with a one dollar general purpose microcontroller, performance analysis showed good results.

SIC^2 is a framework for secure key generation and provisioning on IoT devices using crypto coprocessors. It provides a solution for device manufacturers to write private keys securely into the crypto coprocessor. As a proof of concept, the framework was implemented using an ESP32 and an ATECC608A. It was shown that the ESP32 is vulnerable to format string attacks that can leak cryptographic keys. Storing the keys on the crypto coprocessor closes this security vulnerability, even if the attacker has access to the physical UART port of the ESP32. Additionally, results showed an improvement in TLS handshake time of up to 82%, and energy consumption of the system could be lowered by up to 70% [20].

Although the approach taken by [34] can be implemented more cheaply, performing cryptographic operations and key provisioning locally on the device is still the preferred option as the communication between the smartphone and the IoT device is an additional potential attack vector. The hardware used by [20] is very similar to the hardware used in this thesis, therefore, the expected security benefits are the same. However, using a secure element as the only device hardening technique is not sufficient, since the leakage of cryptographic keys is only one security vulnerability among many. Therefore, the solution proposed in this thesis uses the secure element as only one pillar of a more comprehensive device hardening approach.

3.3 Device Monitoring and Recovery

Research works around device monitoring and recovery can be grouped into two categories: solutions that monitor side channel information and solutions that monitor network traffic. Table 3.3 compares the current device monitoring and recovery solutions.

3.3.1 Side-channel Information Monitoring

In [35], a generally applicable system for detecting malicious behavior and recovering from it is proposed. The system can detect cache side-channel attacks, distributed DoS attacks, and CPU stealing attacks.

The architecture consists of the IoT device and a set of trusted services that are running in the cloud. The IoT device is internally divided into a trusted and an untrusted environment. Malicious activity detection is implemented by a watchdog counter in the trusted part of the IoT device, which needs to be reset by a signal from a trusted service in the cloud. If the counter is not reset, the IoT device will reboot and install a secure version of the firmware to overwrite any malicious application. To detect malicious behavior, a local machine learning model analyzes whether hardware performance counters (low-level data about the performance of the running program) deviate too strongly from the expected behavior. If this is the case, the recovery component is invoked.

Research Work	Solution Name	Devices	Anomaly Detection Method	Monitored Information	Stable State Recovery Method
Zahan et al. [9] (2024)	IoT-AD	Intel NUC mini, various smart plugs	Comparing Event Signatures with ML	Network Traffic	Isolate the device and rollback propagated states
Medwed et al. [35] (2021)	–	Layerscape 1046ARDB	Machine Learning Model	Internal Performance Metrics	Write secure firmware image to device
Wang et al. [8] (2021)	IoT-Praetor	Smart-Things Devices	Pattern Matching with Device Usage Rules	Network Traffic, Smart-Things API	–
Zhang et al. [10] (2018)	HoMonith	Smart-Things Devices	DFA Matching Algorithm	Wireless Network Traffic	–

Table 3.3: Device Monitoring and Recovery Solutions

A limitation of the watchdog counter approach for recovering the device is the lack of knowledge of the IoT device about why it did not get a reset signal. For example, a failure of the trusted services can lead the IoT device to reboot and flash new firmware, although no malicious activity has happened. A similar problem exists with the hardware performance counters used for malicious behavior detection since deviations might not necessarily be a result of malicious activity. Nevertheless, the device is always reset when such a deviation occurs.

3.3.2 Network Traffic Monitoring

There exist three different systems for network traffic monitoring: HoMonith, IoT-Praetor, and IoT-AD. In the following paragraphs, these systems are presented and compared.

HoMonith [10] is a system for detecting malicious behavior of SmartApps, applications that run on Samsung IoT devices. It was designed to detect two types of malicious behaviors: over-privileged accesses (the application uses commands that are not needed for its desired purpose) and event spoofing (the application sends commands to alter the behavior of other SmartApps). The system consists of two main parts: the SmartApp Analysis Module and the Misbehavior Detection Module. The analysis module extracts Deterministic Finite Automatons (DFA) from benign applications by either performing a static analysis of their source code or by analyzing the UI of the application with Natural Language Processing (NLP). The misbehavior detection module then collects all network traffic, filters out the relevant packets, and infers in which state or transition the SmartApp currently is. Then, a DFA matching algorithm is applied to detect whether

the current state of the SmartApp is valid in the automaton of the benign application. To evaluate the system, 30 benign SmartApps were modified to create both an over-privileged and an event-spoofing version. Over 98% of malicious behavior of the modified versions was detected, while only a maximum of 8% of benign application behavior was labeled as malicious. This work was performed using the Samsung SmartThings platform, but the approach of determining DFAs and analyzing the network traffic can be generalized. However, some device manufacturers could modify the wireless traffic patterns, influencing the accuracy of HoMonith.

IoT-Praetor [8] is a system to automatically extract the baseline behavior of IoT devices in the Samsung SmartThings ecosystem and detect malicious application interaction and communication behaviors. To specify the application behavior, a Device Usage Description (DUD) model is proposed. This tool consists of three parts: the Automatic Rule Extraction Module, the DUD Generation Module, and the Behavior Detection Engine. The automatic rule extraction module extracts the interaction behavior of the device by analyzing the applet, applet description, and the device with NLP tools. The communication rules are extracted by obtaining characteristics of network communication from the device. The DUD generation module uses the behavioral information from the automatic rule extraction module to generate Device Usage Rules (DUR) as a whitelist of device behavior. The behavior detection engine analyzes the real-time interaction and communication behavior of the devices. Interaction information is gathered through the SmartThings API which provides data about device location and installed applications. The communication behavior is obtained by monitoring network packets between the devices, the cloud, and the router. The real device behavior is then compared to the DURs with a pattern-matching algorithm to detect any anomalies and take appropriate action such as logging the event or sending an alert. The system successfully detects malicious interaction behavior in 94.5% of the cases and malicious communication behavior in 98% of the cases. The additional delay introduced is in the milliseconds. IoT-Praetor was designed and implemented for the Samsung SmartThings platform, so additional effort is needed to use IoT-Praetor components with other platforms. The authors plan to extend this system to other platforms. Because of the high real-time requirements of IoT devices, the performance of the solution should be further improved. An additional limitation is that IoT-Praetor assumes that the device communication behavior can initially be extracted in a clean environment and does not change in the life cycle of the device.

The IoT Anomaly Detector (IoT-AD) [9] is a framework that detects interaction and communication anomalies of IoT devices and mitigates the effect the effects of these anomalies by providing a method for the devices to recover to their last known stable state. The system is able to detect anomalies that are due to device malfunctions as well as malicious attacks. In IoT-AD, all the traffic from the whole IoT network is routed through the IoT-AD controller. This traffic is then scanned by the event identifier to extract network events. After that, a lightweight machine learning model in the Packet Level Anomaly Detector analyzes these events to detect whether an event has an unknown structure, which would result in discarding it. Then, the Device Interaction Validator asynchronously checks whether the interactions in that event were legitimate. If not, the devices are rolled back to their last known stable state, and the device causing the anomaly is isolated. IoT-AD has an accuracy of up to 98%. The additional delay introduced to the system is around 2 ms. A limitation of IoT-AD is the IoT-AD controller, which is a single

point of failure. It also does not consider all types of failures that could occur in a system, like power failures and is not well scalable to systems with hundreds of IoT devices. The rollback functionality of this system does not flash new firmware onto affected devices but resets internal state variables to the last known secure value (e.g. light on / light off). Also, IoT-AD does not provide the capability to reset the device that is causing the anomaly but only isolates it. Performing anomaly behavior checks synchronously would eliminate the need for rollbacks, however, the Device Interaction Validator would become a performance bottleneck for the whole system.

3.4 Summary

Current research on IoT device hardening emphasizes the importance of proper device configuration. The authors of [5] and [4] show that IoT devices are vulnerable to a variety of attacks when used with their standard configuration. These attacks can be mitigated by configuring the devices properly. In case of higher-end legacy devices whose firmware cannot directly be updated with a more secure device configuration, [3] provides a framework to implant the necessary security measures on the device next to the original firmware. All these papers provide concrete steps that can be taken to harden the devices, however, no concrete implementation of a tool that automates this process is given.

Another important aspect of IoT device security is secure key generation and provisioning. The two main solutions are offloading this functionality to smartphones in the same network [34] or using a crypto coprocessor [20]. It could be shown that both approaches improve security and can for example prevent the leakage of cryptographic keys. This thesis builds directly upon the findings of [20] since almost the same hardware is used. The demonstrated security improvements are therefore assumed to also apply to the solution implemented in this thesis.

The third branch of IoT device hardening research is about device monitoring and recovery. There exists a solution that monitors side-channel information of the devices [35], but with HoMonith [10], IoT-Praetor [8] and IoT-AD [9], the majority of solutions focuses on monitoring the network traffic. These papers use different methods for the detection of malicious behavior, although machine learning approaches are becoming more and more popular. While HoMonith and IoT-Praetor focus purely on the detection of malicious behavior, IoT-AD also automatically isolates the causing device and resets the states of devices that were affected by the anomaly. The solution by [35] is the only one that tries to recover the malicious device itself by writing a secure firmware image to the device after a malicious event occurs. All these monitoring solutions focus on the detection of malicious behavior after the device has already been compromised. None of the solutions extracts the device state and proactively ensures a secure device configuration, which is the problem this thesis is going to address.

Chapter 4

Design

In this chapter, the requirements, architecture, workflow, and components of the ESP-DHS are introduced. The architecture is the foundation for the implementation, which is explained in more detail in Chapter 5. Furthermore, it is elaborated on how the architecture implements the NIST IR 8259A and NIST SP 800-53 security standards.

4.1 Overview

From a high-level perspective, the ESP-DHS ensures the integrity of the ESP32-S3 microcontroller, safeguards it against malicious tampering, and prevents it from unauthorized access. The ESP-DHS consists of two main parts: the firmware running on the ESP32-S3, called "client" and the software running on a physically separate host computer, called "server".

The server requests the security configuration from the client. When the client receives such a request, it reads out the configuration of the microcontroller and sends it back to the server. The configuration consists of security-relevant device properties like the MAC address or model number and a list of security features and whether they are activated or not. If the server recognizes some deactivated security features, these features are automatically activated, and a new version of the client firmware is compiled and flashed onto the ESP32-S3. This workflow and the involved components are described in more detail in Sections 4.3 and 4.4.

4.2 Prerequisites

For ESP-DHS to work properly and to provide a high level of security, several prerequisites need to be satisfied:

- A working Python version 3.9 or higher installation is necessary on the host computer to run the ESP-DHS server.

- The host computer needs to have the ESP integrated development framework (ESP-IDF) installed in order to compile new firmware.

- The server needs access to the ESP32-S3 application source code so that it is able to compile a new firmware version with improved security configurations.

- The host computer needs to have access to a good random key generator so that the secure boot key generated on the server is of high quality and does not pose a security vulnerability by itself.

- The ESP32-S3 and server need to be connected to the Internet or at least be in the same network so that the ESP-DHS client and server are able to communicate.

- The ESP32-S3 needs to be physically connected via USB to the host computer in order to receive firmware updates.

4.3 Components

The ESP-DHS consists of five components: the ESP32-S3 as the client, a secure element that is connected to it, a server program that runs on a host machine, a state database, and a MQTT message broker for the communication between the ESP32-S3 and the server. Figure 4.1 shows an overview of the high-level components of the system, how they are connected, and which protocols are used. The blue annotations show which controls are implemented by the components, which is explained in more detail in Section 4.6.2. In the implementation of the ESP-DHS used for the evaluation, the broker, server, and database reside on the same physical machine, although it would be relatively easy to move them to different machines.

The Python program on the server communicates with the SQLite database via the SQL Alchemy Python library [36]. Between the server and the ESP32-S3 exist two connections, one for exchanging messages and one for flashing new firmware. For exchanging messages, the MQTT protocol is used, therefore, messages always go through the MQTT broker. Communication with the MQTT broker works via a wireless TLS-secured TCP connection. In order to download new firmware, the microcontroller needs to be physically connected to the host computer via its USB UART port. If this connection is not present, the server cannot update the ESP32-S3 and will not do anything besides logging the MQTT message exchange and security state of the ESP32-S3.

As an additional security component, an ATECC608B secure element is physically connected to the ESP32-S3 via the Inter-Integrated Circuit (I2C) protocol. The secure element is intended to improve the security of user applications running on the ESP32-S3. When extracting the device security configuration, the ESP-DHS client checks whether the secure element is connected.

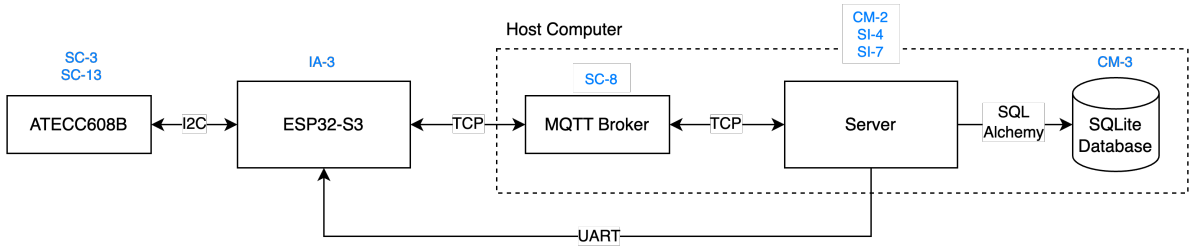


Figure 4.1: Components with their Connections, Protocols, and Standards

4.4 Workflow

The workflow of the ESP-DHS is described in Figure 4.2. The blue annotations show the used protocols and in the case of MQTT, on which topic the message is sent. Since the MQTT broker is only a relay station for the message exchange, it is not modeled in the sequence diagram. Also, the ATECC608B secure element is not modeled, since it is only intended for use in the user application, not the ESP-DHS directly.

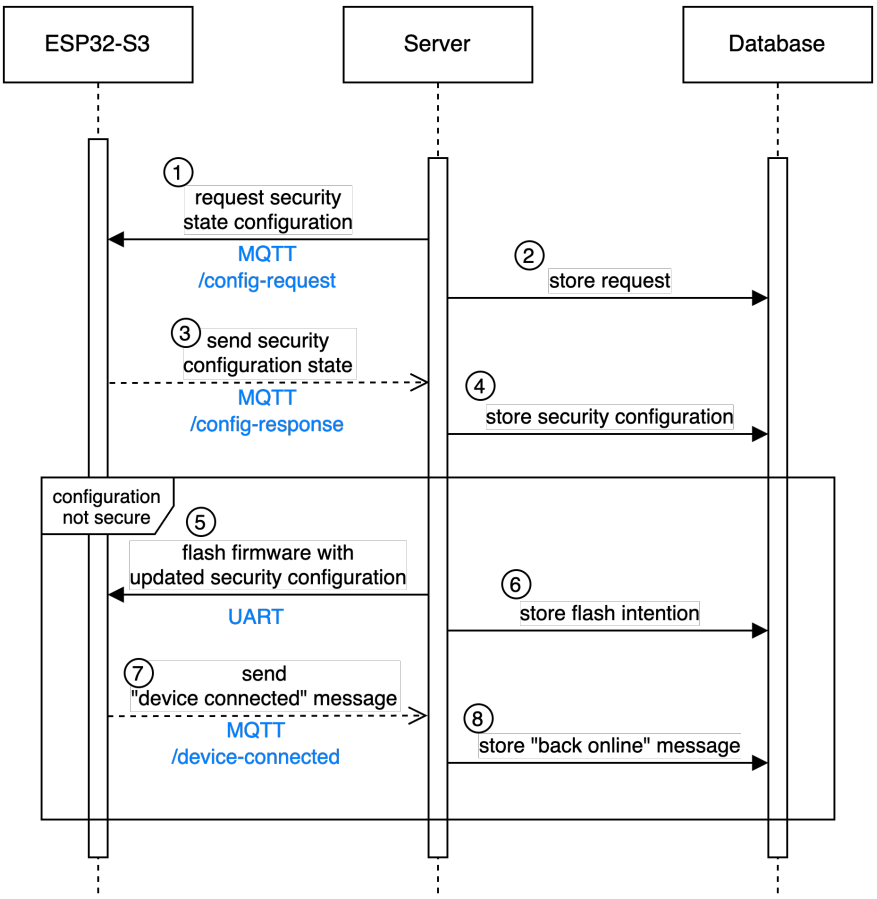


Figure 4.2: ESP-DHS Workflow Sequence Diagram

First, the server requests the security configuration of the ESP32-S3 (1). This configuration includes for example whether secure boot or flash encryption are activated. When the ESP32-S3 receives the state request, it extracts its configuration and sends it back to the server (3). The server then analyzes this configuration to check, whether all the relevant security features are activated. If the configuration is not secure, the server automatically compiles a new version of the firmware with updated security settings and flashes it onto the ESP32-S3 (5). The firmware flashing happens, other than the previous and subsequent message exchanges, over the physical USB UART connection and not wirelessly over MQTT. After being flashed with the new firmware version and a successful reboot, the ESP32-S3 sends a message to the server to notify that it is back online (7). All communication happening between the server and the ESP32-S3 is logged and stored in a database. This includes the security configuration request (2) and the actual configuration that was received (4). It is also stored when the server decides to flash a new firmware onto the client (6) and when the client responds that it is back online after being flashed (8). This data can later be used to reproduce the system communications or detect unexpected changes in the device configuration that were not initiated by the ESP-DHS itself.

4.5 User Applications

The firmware running on the ESP32-S3 is divided into two parts: the ESP-DHS client and the user application (UA). The client is the part of the firmware used for the ESP-DHS itself, this includes establishing the network connection for MQTT messaging or extracting the security configuration. However, the ESP-DHS is only intended to have a supporting role in the IoT system, the user application provides the actual functionality the device is used for. For example, when the ESP32-S3 is used as a weather station, the user application is responsible for reading the data from the sensors and passing it to the IoT gateway, the ESP-DHS client's role is only to make sure the device is always in a proper security state.

In this thesis, a minimal viable example of a user application is implemented to show how the ESP-DHS client, the secure element, and the user application can coexist and interact. The user application lets the built-in LED of the ESP32-S3 blink in random colors. The random values for the red, green, and blue channels of the LED are provided by the ATECC608B secure element.

The firmware on the ESP32-S3 is divided into components, which are shown in Figure 4.3. These are not to be confused with the more high-level ESP-DHS components from Figure 4.1.

The **Main** component is the entry point of the ESP32-S3 firmware. This component is primarily used to initialize other components like the MQTT, Wi-Fi, and ATECC components, but also to start the user application by calling the UA Main component.

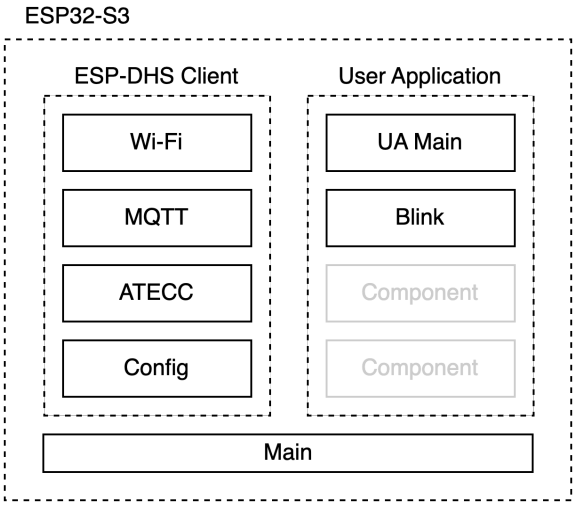


Figure 4.3: ESP-DHS Client and User Application on the ESP32-S3

4.5.1 ESP-DHS Client Components

- The **Wi-Fi** component establishes a connection to a WPA2 secured Wi-Fi network. The network connection can also be used by the user application.
- The **MQTT** component is responsible for managing the MQTT traffic, this includes handling the incoming messages but also sending messages when necessary. This component is set up with the MQTT broker needed for the ESP-DHS and therefore not intended to be used by the user application.
- The **ATECC** component is used for the connection with the ATECC608B secure element. This component provides an interface to get a random number from the secure element. More interface functions could be provided to make this component more useful to the user application.
- The **Config** component extracts the relevant security configuration from the ESP32-S3 and creates a JSON string with the data. This string can then be sent back to the server by the MQTT component.

4.5.2 User Application Components

A user application could potentially consist of as many components as necessary, in Figure 4.3 this is shown with two gray boxes in the user application. The example user application only consists of two components.

- The **UA Main** component is the entry point of the user application. It initializes the blink component, gets a random number from the ATECC608B, and loops through the random digits to calculate the color to be shown.

- The **Blink** component sets up the connection with the LED and lets it glow in the color received by the UA Main component.

4.6 Security Standards

The ESP-DHS implements security controls from the NIST IR 8259A and NIST SP 800-53 standards. This section shows, how these standards are implemented in the ESP-DHS. NIST IR 8259A is presented in more detail in Section 2.6.2, NIST SP 800-53 in Section 2.6.3.

4.6.1 NIST IR 8259A Mapping to ESP32-S3 Components

Since the ESP32-S3 microcontroller is used in the ESP-DHS, the solution adheres to the NIST IR 8259A standard. This is because the NIST IR 9259A standard is a subset of the PSA standard, and the ESP32-S3 is PSA certified [28]. NIST IR 9259A consists of seven capabilities IoT devices should have. The following paragraphs show, how these capabilities are implemented in the microcontroller. A summary of the capabilities is provided in Table 2.3, the implementations are listed in Table 4.1.

The first device capability is **Device Identification**, which means that every device needs to be uniquely identified both physically and logically [6]. The factory MAC address of the ESP32-S3 uniquely identifies the microcontroller physically. In case the user defines a custom MAC address, he is responsible for ensuring uniqueness [37]. Logical identification, for example, when connecting the device to a remote endpoint, can be ensured with digital signatures. The ESP32-S3 provides a dedicated digital signature component [16].

The **Device Configuration** capability states that the device's software configuration can be changed, but only by authorized entities. Changing the configuration requires flashing a new firmware onto the microcontroller that has implemented these changes. By using the secure boot feature of the ESP32-S3, it can be ensured that only authorized entities that have the secure boot signing key can flash new firmware [18].

The **Data Protection** capability ensures that devices protect stored and transmitted data from unauthorized access. Data stored on the ESP32-S3 can be encrypted with flash encryption. To protect transmitted data, security protocols like WPA3 and HTTPS are supported [38].

The **Logical Access to Interfaces** capability ensures the device's local and network interfaces are only accessible by authorized entities. This is ensured by secure boot, which makes sure that no unauthorized code can run on the device that could access the device's APIs [18].

The **Software Update** capability makes sure that authorized entities can update the software or firmware running on the device by a secure and configurable mechanism. The ESP32-S3 provides two main mechanisms for updating its firmware: over-the-air (OTA)

Capability	Implementation
Device Identification	Physical identification possible by factory MAC address [37], logical identification for remote connection by digital signature support [16]
Device Configuration	Project configuration is changeable but requires flashing new firmware [24]
Data Protection	Flash encryption protects stored data [24], WPA3 and HTTPS protect (wirelessly) transmitted data [38]
Logical Access to Interfaces	Secure boot prevents unauthorized code from running that could access APIs [18]
Software Update	Updates can be distributed OTA or via UART, secure boot ensures the integrity of the flashed firmware [18]
Cybersecurity State Awareness	Secure boot key revocations are stored in eFuses [18].

Table 4.1: NIST IR 8259A [6] Capabilities with ESP32-S3 Implementation

updates and standard firmware updates via UART. Secure boot can be used with both methods and ensures the integrity of the firmware [18].

The **Cybersecurity State Awareness** capability ensures that the device can keep track of its security state and report it to authorized entities. If the verification of an app image fails during the boot process, the corresponding key is revoked [18]. This information is burned into eFuses, which cannot be changed anymore but are still possible to read out from application code or a connected device [17].

Many of these capabilities are implemented by secure boot and flash encryption. However, since these features are not enabled by default, it is crucial for the device security to activate them properly.

4.6.2 NIST SP 800-53 Mapping to ESP-DHS Components

The ESP-DHS implements several controls from the NIST SP 800-53 standard. The following paragraphs describe which controls are implemented by which component, the control IDs are indicated inside brackets in the text. A summary of the controls with their corresponding implementation is found in Table 4.2. The components in Figure 4.1 are also annotated with the corresponding controls in blue color.

On the server, a baseline configuration is defined, which outlines the security features that must be activated to ensure a secure system. The ESP-DHS automatically enables these features on the ESP32-S3 if they are deactivated (CM-2). The server requests the security configuration at regular intervals (SI-4), keeps track of any changes, and stores them in a database (CM-3).

ID	Title	Component	Implementation
CM-2	Baseline Configuration	Server	The server has a predefined secure configuration that is automatically applied to the ESP32-S3.
CM-3	Configuration Change Control	Database	All changes to the ESP32-S3 configuration are stored and documented in the database alongside all necessary metadata.
SC-3	Security Function Isolation	ATECC608B	The secure element is responsible for secure cryptographic operations and key storage and is physically separated from the rest of the microcontroller.
SC-8	Transmission Confidentiality and Integrity	MQTT Broker	Communication between the ESP-DHS server and client happens TLS encrypted via the MQTT broker.
SC-13	Cryptographic Protection	ATECC608B, ESP32-S3	The ATECC608B provides cryptographic functions to protect data in the user program. Flash encryption and secure boot cryptographically protect the microcontroller's memory and firmware [16].
SI-4	System Monitoring	Server	The server continuously requests a configuration state update of the ESP32-S3 and stores it in the database.
SI-7	Software, Firmware, and Information Integrity	ESP32-S3	The secure boot feature of the ESP32-S3 prevents unauthorized firmware from running on the device [18].
IA-3	Device Identification and Authentication	ESP32-S3	The ESP32-S3 provides unique MAC addresses. It can establish secure connections via TLS to remote endpoints due to its digital signature peripheral [16].

Table 4.2: Implementation of NIST SP 800-53 Controls in ESP-DHS.

Communication between the server and the client is routed through the MQTT broker. The broker is TLS encrypted, therefore, the confidentiality and integrity of the exchanged messages can be guaranteed (SC-8).

To ensure the integrity of the firmware running on the device, the ESP32-S3 employs secure boot. This prevents unauthorized code from running on the microcontroller (SI-7). The ESP32-S3 is uniquely identified by its factory-assigned MAC address. Furthermore, it can establish a secure device identity for remote connections due to its digital signature capabilities (IA-3). The ESP32-S3 uses cryptographic protection in multiple instances: flash encryption and secure boot sign the flash memory and firmware cryptographically to prevent malicious parties from reading it out. The ATECC608B secure element provides cryptographic functions to enhance security in the user application (SC-13). The secure element also makes sure that security functions are isolated from non-security functions, by providing a secure interface for cryptographic operations and storage for cryptographic keys (SC-3).

4.7 Summary

This chapter provided a high-level overview of the structure and functionality of the proposed ESP-DHS. The system consists of five components: the ATECC608B secure element, ESP32-S3, MQTT broker, Python server, and SQLite database. The firmware running on the ESP32-S3 is divided into two parts: the ESP-DHS client and the user application. The ESP-DHS client has a supporting role and ensures the microcontroller is always in a secure state, the user application implements the actual functionality of the device. The Python server is responsible for checking whether the configuration of the microcontroller is secure and updating the firmware if necessary. There exist two connections between the ESP32-S3 and the server, one is over Wi-Fi and uses the MQTT protocol to exchange messages, and the other is a physical USB connection and used for flashing firmware onto the microcontroller. The workflow of the ESP-DHS is as follows: the server requests a security configuration of the ESP32-S3, this configuration contains, for example, whether secure boot, flash encryption, or memory protection are activated. The ESP-DHS client extracts this configuration from the microcontroller and sends it back to the server, where it is evaluated. If any deactivated security features are detected, the server automatically updates and compiles a new version of the client firmware and flashes it onto the ESP32-S3. The microcontroller sends a message back to the server as soon as it has booted the new firmware. The server stores all messages and device configurations in a database, this data can be used to keep track of exchanged messages and changes in the security configuration.

Since the ESP32-S3 microcontroller is used in the ESP-DHS, the NIST IR 8259A standard is already implemented. The ESP32-S3 implements most of the required capabilities by flash encryption and secure boot, which are not activated by default. The system also implements selected controls of the NIST SP 800-53 standard.

Chapter 5

Implementation

In this chapter, the implementation of the ESP-DHS is explained, every section describes a particular component. The source code for the ESP-DHS server [39] and the client firmware [40] can be found on GitHub. A detailed description of how to set up the project is in Appendix A.

5.1 Server

The server is responsible for requesting the security configuration from the ESP32-S3, evaluating it, compiling new firmware if necessary, and flashing it to the microcontroller. In this section, the file structure and selected code snippets of the server project are described. The programming language used for the server implementation is Python version 3.9.6 [41]. Functions with a leading underscore are intended for file-internal use only.

5.1.1 Overview

The `ESP-DHS-server` folder contains all files necessary for the server to operate. The following list explains the purpose of all the contained files and folders of the server, the directory tree is shown in Figure 5.1.

- The `backups/` folder contains copies of secure boot signing keys that were found in the client project. The location of this folder does not necessarily need to be inside the ESP-DHS-server folder and can be set manually in the .env file.
- The `database.sqlite3` file is where all the data from the database is stored. The location of this file can also be set manually in the .env file.
- The `ca.crt` file is the TLS certificate used for the secured connection with the MQTT broker.

- The `.env` file contains environment variables for both the server and the client. This file should be kept secret. A detailed explanation of all environment variables is provided in Section 5.1.2.

- The `.env.example` file is the blueprint for the .env file, it is intended to be shared to provide the structure of the .env file and does not contain sensitive information. It also contains explanations for the environment variables.

- The `main.py` file is the entry point of the server application. It contains the main program loop that sends the period configuration requests and the callbacks to handle incoming MQTT messages. The evaluation of the ESP32-S3 security configuration also happens in this file.

- The `mqtt_client.py` file contains the logic for setting up the Paho MQTT client.

- The `compile.py` file contains the logic used for activating security features, compiling the firmware, and flashing it onto the ESP32-S3.

- The `database.py` file is used for connecting the server to a database using SQLAlchemy and contains a helper function to add entries to the database more easily.

- The `models.py` contains the schema for the database tables that are used for storing new entries in it.

- The `run_with_env.sh` file is necessary to run ESP-IDF terminal commands from the compile.py Python script. The IDF export file is not called directly but through this shell script. This is explained in more detail in Section 5.1.6.

- The `README.md` file contains instructions on how to set up the ESP-DHS server.

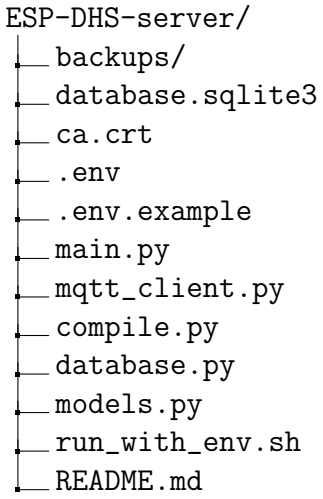


Figure 5.1: Project File Structure of the Server

5.1.2 Environment Variables

Environment variables are stored in the `.env` file and used to separate project settings and security parameters from the code. The variables are imported into the Python code. In order to keep sensitive information like the MQTT broker username and password confidential, the `.env` file should not be made public, for example by adding it to a Git repository. However, the `.env.example` file has the same structure as the `.env` file, but with dummy values and can be published. It can then be used as a blueprint when for creating the actual `.env` file. Listing 5.1 shows an example of a `.env` file, the following list explains what the variables are used for. The variables can be used in Python code with the syntax `os.getenv('VARIABLE_NAME')`, sometimes this expression is assigned to a constant so that `VARIABLE_NAME` can be used directly.

- The `HOST` variable is the Fully Qualified Domain Name (FQDN) of the device on which the MQTT broker resides. This domain name is used to locate the MQTT broker so that messages can be exchanged with the client.
- The `USERNAME` variable is the required username for authentication with the MQTT broker.
- The `PASSWORD` variable is the required password for authentication with the MQTT broker.
- The `DB_PATH` variable is the path to the SQLite 3 database file.
- The `PROJECT_PATH` variable is the location to the root directory of the ESP32-S3 firmware source code.
- The `ESP_IDF_PATH` variable is the location of the root directory of the Espressif Integrated Development Framework (ESP-IDF).
- The `BACKUP_DIR_PATH` variable is the path to the folder where the secure boot key backups will be stored.
- The `KEY_FILE_NAME` variable is the name of the secure boot signing key, including the file extension.
- The `PORT` variable is the name of the USB port at which the ESP32-S3 is connected to the host computer.
- The `TARGET` variable is the architecture for which the firmware will be compiled, i.e. which type of ESP microcontroller is used.
- The `BAUD_RATE` variable contains the serial baud rate used for flashing firmware to the ESP32-S3.
- The `REQUEST_INTERVAL` variable is the interval in seconds in which the security configuration of the ESP32-S3 is requested. If the ATECC608B is connected to the ESP32-S3, this value should be at least 60, because when the secure element gets disconnected, the response of the client takes longer. This is explained in more detail in Section 6.2.1.

```
# .env

############## MQTT
HOST = 'macbook-air-von-michel.home'
USERNAME = 'user1'
PASSWORD = 123456

############## DATABASE
DB_PATH = '/database.sqlite3'

############## ESP-IDF
PROJECT_PATH = '/Users/michel/esp/v5.2.1/projects/BA-project'
ESP_IDF_PATH = '/Users/michel/esp/v5.2.1/esp-idf'
BACKUP_DIR_PATH = '/Users/michel/BA-project-server/backups'
KEY_FILE_NAME = 'secure_boot_signing_key.pem'
PORT = '/dev/cu.usbserial-140'
TARGET = 'esp32s3'
BAUD_RATE = 115200

############## REQUEST INTERVAL
REQUEST_INTERVAL = 60
```

Listing 5.1: Example of a Server `.env` File

5.1.3 Main Loop

The ESP-DHS server can be started by running the `main.py` file. Listing 5.2 shows the code of the main loop. The `run_main_loop` variable is used to control when configuration requests can be sent. The variable is initially set to true, however, before new firmware is compiled and flashed, it is set to false. This prevents sending unnecessary requests since the ESP32-S3 cannot respond during flashing. The `client` variable is the MQTT client object, when the `loop_start()` method is called, it is ready to send and receive messages. Inside the main loop, the configuration request messages are sent to the topic `/config-request`. At the end of every loop iteration, the program waits for the in the `REQUEST_INTERVAL` environment variable specified number of seconds before continuing.

```
# main.py

run_main_loop = True
client = init_mqtt_client()
client.loop_start()
while True:
  if run_main_loop:
    publish_message(client, '/config-request')
    time.sleep(int(os.getenv('REQUEST_INTERVAL')))
```

Listing 5.2: Server Main Loop

5.1.4 MQTT Client

The MQTT client provides functionality for publishing messages and connecting to the broker. The functionality of MQTT and the broker is explained in more detail in Section 5.4. To establish the MQTT connection, the Paho Python package [42] is used. Listing 5.3 shows an excerpt of the MQTT client code.

The function `init_mqtt_client()` first creates the Paho client, sets the TLS certificate, sets the username and password for the broker, and finally connects to it. A number of callback functions are set that are executed when specific events occur. For example, when the connection to the broker was successful, the `_on_connect()` function is executed, in which the client gets subscribed to topics. Other callback functions are defined for when a topic subscription was successful or a message was received or published. These functions are only used for logging the events to the console.

The server sends the configuration request to the `/config-request` topic, and the ESP-DHS client responds with the payload on the `/config-response` topic. Directly after being restarted, the client also publishes its configuration to the `/device-connected` topic. This flow is shown in Figure 4.2.

The `publish_message()` function is a helper function that first publishes a message to the provided topic. The message is then stored in the database.

```
# mqtt_client.py

import paho.mqtt.client as paho

def _on_connect(client, userdata, flags, rc):
  if rc == 0:
    print('connection successful')
    client.subscribe('/config-response')
    client.subscribe('/device-connected')

def publish_message(client, topic, payload=None):
  return_code, message_id = client.publish(topic, payload)
  database.add_row(Message(topic=topic, status=return_code, type='sent')
   )

def init_mqtt_client():
  client = paho.Client()
  client.tls_set('./ca.crt')
  client.username_pw_set(os.getenv('USERNAME'), os.getenv('PASSWORD'))
  client.on_connect = _on_connect
  client.on_publish = _on_publish
  client.on_subscribe = _on_subscribe
  client.on_message = _on_message
  client.connect(os.getenv('HOST'), 8883)
  return client
```

Listing 5.3: Excerpt of the MQTT Client Code

5.1.5 Configuration Evaluation

The security configuration is evaluated after a message is received on the `/config-response` topic. Three security features are evaluated and activated if necessary: secure boot, flash encryption, and memory protection. More details about these features and how they enhance device security can be found in Section 2.2.

Listing 5.4 shows the code that performs the evaluation. The `@client.topic_callback` decorator signifies that the `handle_config_response` function contains the logic for handling received messages on the specified topic. The message payload contains the current security configuration of the ESP32-S3 in JSON format as shown in Listing 5.19 and is stored in the `configuration` variable. Before the evaluation begins, the message and configuration are stored in the database. For the evaluation, it is checked whether the keys of the three previously mentioned security features are set to false. If so, the feature needs to be activated and added to the `features` array. If the `features` array is empty, execution stops here. If not, the main loop is stopped and an entry is stored in the `Flashes` table of the database with "pending" in the `status` column and the current timestamp in the `start` column. The entry also contains which features will be activated. After that, the `features` array is passed to the `compile_secure` function, which compiles the ESP32-S3 firmware with updated security features and flashes it to the device. If the function returns without raising an exception, the status of the database entry is updated to "success", the current timestamp is added to the `end` column, and the main loop is resumed.

```
# main.py

run_main_loop = True

@client.topic_callback('/config-response')
def handle_config_response(client, userdata, message):
  global run_main_loop
  print('/config-response message received')
  configuration = json.loads(message.payload)
  compare_configurations(configuration)
  message_db = Message(topic=message.topic, type='received')
  configuration_db = Configuration(message=message_db, **configuration)
  database.add_row([message_db, configuration_db])
  features = []
  if configuration['flash_encryption_enabled'] == False:
    features.append('flashencryption')
  if configuration['secure_boot_enabled'] == False:
    features.append('secureboot')
  if configuration['memory_protection_enabled'] == False:
    features.append('memoryprotection')
  if features:
    print('The following features will be activated:', features)
    run_main_loop = False
    flash_db = Flash(status='pending',
                     flashencryption='flashencryption' in features,
                     secureboot='secureboot' in features,
                     memoryprotection='memoryprotection' in features)
    database.add_row(flash_db)
```

```
29      try:
30        compile_secure(features)
31        flash_db.status = 'success'
32      except Exception as e:
33        print('Could not flash firmware:', e, '\nYou may need to restart
    the device manually.')
34        flash_db.status = 'error'
35      flash_db.end = datetime.now(timezone.utc).replace(microsecond=0)
36      database.add_row(flash_db)
37      run_main_loop = True
```

Listing 5.4: Security State Evaluation

The `compare_configurations` function shown in Listing 5.5 is used to log detected changes to the console. The global `current_config` dictionary holds the last known security configuration of the ESP32-S3. Received configurations are compared with the values in this dictionary and differences are logged to the console. At the end, `current_config` gets replaced with the new configuration. An example of a resulting output is shown in Listing 5.6.

```
1  # main.py
2
3  current_config = {}
4
5  def compare_configurations(new_config):
6    global current_config
7    if current_config != {}:
8      change_detected = False
9      output = ""
10     for key in current_config:
11       if current_config[key] != new_config[key]:
12         change_detected = True
13         output += key + ': ' + str(current_config[key]) + ' -> ' + str(
     new_config[key]) + '\n'
14     if change_detected:
15       print('\nChange in config detected:')
16       print(output)
17   current_config = new_config
```

Listing 5.5: Compare and Print Configuration Differences

```
1  Change in config detected:
2  flash_encryption_enabled: False -> True
3  flash_encryption_mode: 0 -> 1
4  secure_boot_enabled: False -> True
5  memory_protection_enabled: False -> True
6  memory_protection_locked: False -> True
```

Listing 5.6: Example Console Output of the `compare_configurations` Function

5.1.6 Compiling and Flashing Firmware

When deactivated security features are detected as described in Section 5.1.5, these security features get activated in the source code of the ESP32-S3 firmware, which gets compiled and then flashed onto the microcontroller. This requires that the source code of the firmware is on the same machine as the ESP-DHS server and accessible from the Python code. Details about the ESP-IDF project that contains the source code of the firmware can be found in Section 5.3.

To edit the client project configuration from Python code, the Kconfiglib package [25] is used. Erasing and restarting the ESP32-S3 is done with the esptool package [43]. The firmware is built using the `idf.py` command line tool from the ESP-IDF. Flashing firmware onto the microcontroller can be done either with esptool or ESP-IDF.

Initializing IDF Commands

Since the ESP-IDF does not provide an API for using it programmatically, a command line needs to be simulated in a subprocess. Listing 5.7 shows how this is implemented. To prepare the command line for running the `idf.py` utility, the IDF `export.sh` script needs to be executed. However, doing this directly leads to an error because the script modifies environment variables, that are outside its scope when the subprocess spawns a new shell. Therefore, the path to `export.sh` is passed to the wrapper script `run_with_env.sh`, which is executed. After that, the shell is ready to run `idf.py`. The `-project-dir` flag specifies the location of the client project, and the `-port` flag the USB port on which the ESP32-S3 is connected. These commands are stored in the `init_commands` tuple and need to be prepended every time `idf.py` functionality is used.

```
# compile.py

init_commands = (
  './run_with_env.sh', ESP_IDF_PATH + '/export.sh',
  'idf.py',
  '--project-dir', PROJECT_PATH, '--port', PORT
)
```

Listing 5.7: Initializing Commands for `idf.py`

Running IDF Commands

The `_run_commands()` function shown in Listing 5.8 is a utility for running a list of commands received as an argument in a subprocess. The `while`- and `for`-loops are for printing the output to the command line in real time.

```python
# compile.py

def _run_commands(commands):
  process = subprocess.Popen(commands, stdout=subprocess.PIPE, stderr=
    subprocess.PIPE, text=True)
  while True:
    output = process.stdout.readline()
    if output == '' and process.poll() is not None:
      break
    if output:
      print(output.strip())
  for line in process.stdout:
    print(line.strip())
```

Listing 5.8: Run Command Utility Function

Erasing Flash Memory

Listing 5.9 shows how the esptool API is used to erase the flash memory of the ESP32-S3. This is done before flashing new firmware. Although the flash command overwrites old firmware, in rare cases errors might occur when the flash was not erased completely beforehand. Esptool commands are provided to the `esptool.main()` function in a list. Since esptool is used under the hood of ESP-IDF, esptool commands could also be used through `idf.py`. However, using it directly is preferable since the logic can be accessed directly through the API and no command line needs to be simulated in a subprocess.

```python
# compile.py

def _erase_flash():
  command = (
    '--chip', TARGET,
    f'--port={PORT}',
    'erase_flash')
  esptool.main(command)
```

Listing 5.9: Erase Flash Memory of the ESP32-S3

Changing Security Features

The `_change_security_features()` function shown in Listing 5.10 is responsible for changing the client project configuration. When the `mode` argument is "activate", the security features provided in the `features` argument are activated. If the value is "deactivate", the `features` argument is ignored and all security features are deactivated. This is only used for debugging. The `kconfig` variable holds a Kconfig object. This object keeps track of all possible configurations, options, and default values that were defined in the root Kconfig file. Since ESP-IDF projects do not use a single Kconfig file but multiple hierarchical ones, it is necessary to set some environment variables that allow kconfiglib

to search the project for all the necessary files. With `kconfig.load_config()`, the sdkconfig file from the client project with the actual values is loaded and can thereafter be manipulated with the `set_value()` method.

```
# compile.py

import kconfiglib

def _change_security_features(mode: Literal['activate', 'deactivate'],
    features = None):
  if mode == 'activate' and features == None:
    raise Exception('Cannot activate features if none are listed.')

  kconfig_path = ESP_IDF_PATH + '/Kconfig'
  sdkconfig_path = PROJECT_PATH + '/sdkconfig'

  if not os.path.exists(sdkconfig_path):
    print('Create the sdkconfig file at', sdkconfig_path)
    open(sdkconfig_path, 'w')

  os.environ['COMPONENT_KCONFIGS_PROJBUILD_SOURCE_FILE'] = PROJECT_PATH
    + '/build/kconfigs_projbuild.in'
  os.environ['COMPONENT_KCONFIGS_SOURCE_FILE'] = PROJECT_PATH + '/build/
    kconfigs.in'
  os.environ['IDF_PATH'] = ESP_IDF_PATH
  os.environ['IDF_TARGET'] = TARGET

  kconfig = kconfiglib.Kconfig(kconfig_path)
  kconfig.load_config(sdkconfig_path)
  _set_debug_sdkconfig(kconfig)

  if mode == 'activate':
    if 'secureboot' in features:
      kconfig.syms['SECURE_BOOT'].set_value('y')
    if 'flashencryption' in features:
      kconfig.syms['SECURE_FLASH_ENC_ENABLED'].set_value('y')
    if 'memoryprotection' in features:
      kconfig.syms['ESP_SYSTEM_MEMPROT_FEATURE'].set_value('y')
  elif mode == 'deactivate':
    kconfig.syms['SECURE_BOOT']                .set_value('n')
    kconfig.syms['SECURE_FLASH_ENC_ENABLED']   .set_value('n')
    kconfig.syms['ESP_SYSTEM_MEMPROT_FEATURE'].set_value('n')
  else:
    raise TypeError('Invalid value for the mode variable.')

  kconfig.write_config(sdkconfig_path)
```

Listing 5.10: Change Client Project Security Features

Setting Debug Options

Activating secure boot and flash encryption burns eFuses. Therefore, these features cannot be turned off once activated, which poses a problem for debugging the application. To

keep the ability to turn off security features, eFuses need to be simulated in flash. This requires setting some specific values in the client project configuration, which is explained in more detail in Section 5.3.2. The `_set_debug_sdkconfig()` function shown in Listing 5.11 activates these debugging options. It is very important to remove the function call to `_set_debug_sdkconfig()` from `compile_secure()` once the system is used in production since simulated eFuses do not provide any security.

```
# compile.py

def _set_debug_sdkconfig(kconfig):
  kconfig.syms['PARTITION_TABLE_CUSTOM'].set_value('y')
  kconfig.syms['PARTITION_TABLE_OFFSET'].set_value('0x10000')
  kconfig.syms['EFUSE_VIRTUAL'].set_value('y')
  kconfig.syms['EFUSE_VIRTUAL_KEEP_IN_FLASH'].set_value('y')
```

Listing 5.11: Activate Debug Configuration

Generate Secure Boot Key

When secure boot needs to be activated, a new signing key is automatically generated using the Python cryptography module. The standard RSA key size of Secure Boot V2 is 3072 bits. If a key file already exists in the client project, it is copied to a backup folder before it is overwritten with the new key. Listing 5.12 shows the process of backing up the old key, generating a new one, and copying it to the client project folder.

```
# compile.py

from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization

def _generate_signing_key():
  key_file_path = PROJECT_PATH + '/' + KEY_FILE_NAME
  # backup old key if exists
  if os.path.exists(key_file_path):
    current_datetime = datetime.datetime.now().strftime('%Y-%m-%d_%H:%M
   :%S')
    backup_key_name = 'backup_' + current_datetime + '_' + KEY_FILE_NAME
    shutil.copyfile(key_file_path, BACKUP_DIR_PATH + '/' +
   backup_key_name)
  # create new key
  private_key = rsa.generate_private_key(public_exponent=65537, key_size
   =3072)
  private_key_pem = private_key.private_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PrivateFormat.TraditionalOpenSSL,
    encryption_algorithm=serialization.NoEncryption()
  )
  with open(key_file_path, 'w') as key_file:
    key_file.write(private_key_pem.decode())
```

Listing 5.12: Secure Boot Signing Key Generation

Compile Secure Firmware

Inside `handle_config_response()`, the `compile_secure()` function from Listing 5.13 is called. The `features` argument contains the security features that need to be activated. First, the target architecture for the compilation is set. If secure boot needs to be activated, the signing key is generated and copied to the client project. Then, the security features are activated in the project configuration by calling `_change_security_features()`. This only works if the client project was built at least once manually and the `build/` directory is not empty. After that, the bootloader is compiled and flashed before the rest of the application image is flashed. These two steps must be done separately after secure boot is activated. Because some of the steps include file manipulations, it is necessary to wait a second between function calls with `time.sleep()` so that the file system is stable before flashing the binaries. Finally, the ESP32-S3 is restarted.

In the compile.py file, there exists the function `compile_standard()`, which is only intended for debugging purposes. It compiles and flashes the firmware with deactivated security features so that the ESP32-S3 can be quickly made ready for testing the ESP-DHS functionality.

```
# compile.py

def compile_secure(features = None):
  os.environ['IDF_TARGET'] = TARGET
  if 'secureboot' in features:
    _generate_signing_key()
  _erase_flash()
  _change_security_features('activate', features)
  time.sleep(1)   # necessary so that the right files are flashed
  commands = init_commands + (
    'fullclean',  # delete the build directory
    'bootloader', # compile the bootloader
  )
  _run_commands(commands)
  time.sleep(1)
  _flash_bootloader()
  time.sleep(1)
  commands = init_commands + (
    'flash', # build and flash the partition table and app image
  )
  _run_commands(commands)
  _restart_application()  # manually restart the ESP32
  print('Finished secure compiling and flashing.')
```

Listing 5.13: Compile and Flash Secure Firmware

5.1.7 Device Reconnection

When the ESP32-S3 is restarted, for example on the first boot or after being flashed with new firmware, it sends a message to the `/device-connected` topic with its current security configuration in the payload. The server calls the `compare_configurations()` function described in Section 5.1.5 to log detected changes to the console. The message and configuration are stored in the database. The implementation is shown in Listing 5.14.

```
# main.py

@client.topic_callback('/device-connected')
def handle_device_start(client, userdata, message):
  print('/device-connected message received')
  configuration = json.loads(message.payload)
  compare_configurations(configuration)
  message_db = Message(topic=message.topic, type='received')
  configuration_db = Configuration(message=message_db, **configuration)
  database.add_row([message_db, configuration_db])
```

Listing 5.14: Device Reconnection Handling

5.1.8 Database Connection and Manipulation

The SQLite database is accessed by the Python code using the SQLAlchemy package [36]. The package facilitates the database connection but also provides object-relational mapping (ORM) functionalities. By using SQLAlchemy instead of the built-in SQLite package, the database schema is kept in sync with the models used in the code. It also facilitates switching from SQLite to another type of database like MySQL at a later point.

Listing 5.15 shows the code responsible for database communications. The `engine` variable holds the SQLAlchemy object with the pointer to the database file. If the file does not yet exist, it will be created by `create_engine()`. Rows are committed to the database using the `add_row` utility function. The argument can either be a single Object of an ORM model or a list of models to commit multiple entries at once. The available ORM models are explained in Section 5.2.2.

```python
# database.py

from sqlalchemy import create_engine
from sqlalchemy.orm import Session

engine = create_engine('sqlite://' + os.getenv('DB_PATH'))
Base.metadata.create_all(engine)

def add_row(rows):
  with Session(engine) as session:
    if type(rows) in [list, tuple]:
      session.add_all(rows)
    elif isinstance(rows, object):
      session.add(rows)
    else:
      raise Exception('Parameter type not valid.')
    session.commit()
```

Listing 5.15: Database Connection and Manipulation

5.2 Database

The server stores messages, configurations, and flashes in the database so that the state of the system and changes in the device configuration can be tracked. The database architecture used is SQLite 3 [44], since it provides the convenience of a relational database while being easy to set up.

5.2.1 Structure

The database consists of three tables: messages, flashes, and configurations. The entity-relationship diagram in Figure 5.2 shows the relationships between the tables and their columns with data types.

Messages Table

The messages table stores all sent and received MQTT messages and has the following columns:

- `message_id` is the primary key of the relation.
- `created_at` is the timestamp when the database entry was created. This time should be approximately the same as the time the message was sent or received.
- `topic` is the MQTT topic on which the message was sent or received.

- `status` contains the status code of the message. For received messages, the status code is set to `NULL`. A status code of `0` for sent messages means success.
- `type` contains the value "sent" for sent messages and "received" for received messages.

Configurations Table

The configurations table contains the security configurations of the ESP32-S3 that were received by MQTT messages. Every configuration references the message with which it was received. Since a message contains exactly one or zero configurations, the messages and configurations have an optional one-to-one relationship. The following list explains the columns of the configurations table.

- `configuration_id` is the primary key of the relation.
- `message_id` is a foreign key that references the primary key of the messages table.
- All other columns store the values of the JSON security configuration, which is explained in Section 5.3.4.

Flashes Table

The flashes table stores an entry every time a new firmware is flashed to the microcontroller. The entry is committed to the database before compilation begins and updated once the firmware is flashed successfully.

- `flash_id` is the primary key of the relation.
- `start` contains the timestamp before the compilation started.
- `end` contains the timestamp after the firmware was flashed successfully.
- `status` is initially "pending", but gets updated to "error" or "success" after the flash.
- `flashencryption` is `1` when the feature got activated, `0` otherwise.
- `secureboot` is `1` when the feature got activated, `0` otherwise.
- `flashencryption` is `1` when the feature got activated, `0` otherwise.

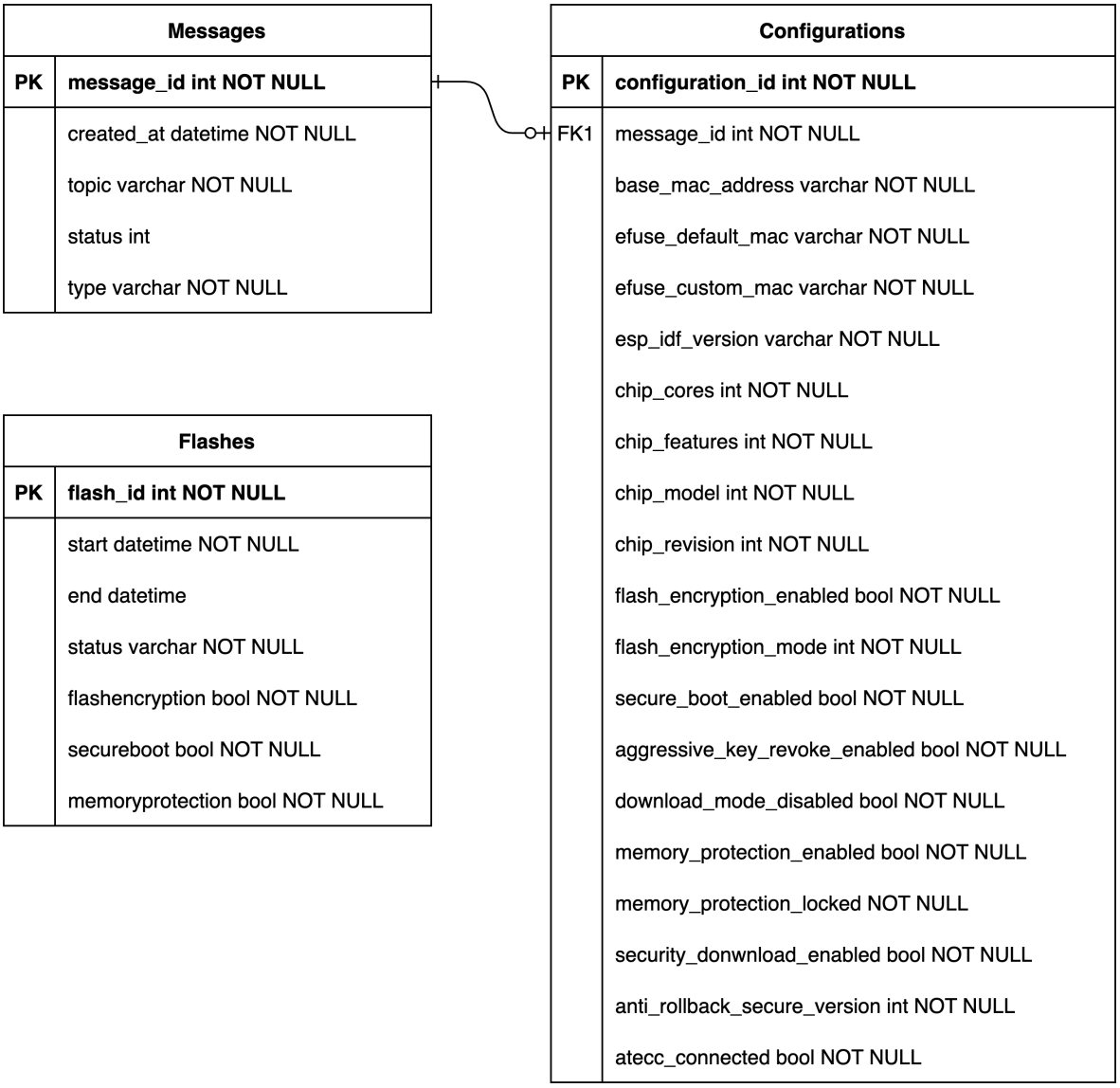


Figure 5.2: Entity-relationship Diagram of the Database

5.2.2 ORM Models

The SQLAlchemy [36] ORM is used to represent database models in the server Python code, the used models are shown in Listing 5.16. Models are created with classes that inherit an ORM base class. The `__tablename__` property specifies the name that will be used in the database. Columns are defined as properties of type `Mapped`, in square brackets is the datatype provided. A datatype wrapped in an `Optional` means that the column can have `NULL` values. If necessary, more configuration about a column can be provided by assigning the `mapped_column()` function, for example, to specify that a column stores the primary key. Relations between models can be defined using the `relationship()` function.

```python
# models.py

class Message(Base):
  __tablename__ = 'messages'
  message_id:    Mapped[int] = mapped_column(primary_key=True)
  created_at:    Mapped[datetime] = mapped_column(server_default=func.
   now())
  topic:         Mapped[str]
  status:        Mapped[Optional[int]]
  type:          Mapped[str]
  configuration: Mapped['Configuration'] = relationship(back_populates='
   message')

class Configuration(Base):
  __tablename__ = 'configurations'
  configuration_id: Mapped[int] = mapped_column(primary_key=True)
  message_id:       Mapped[int] = mapped_column(ForeignKey('messages.
   message_id'), unique=True)
  message:          Mapped['Message'] = relationship(back_populates='
   configuration', single_parent=True)
  # ESP32-S3 security configuration fields omitted

class Flash(Base):
  __tablename__ = 'flashes'
  flash_id:         Mapped[int] = mapped_column(primary_key=True)
  start:            Mapped[datetime] = mapped_column(server_default=func
   .now())
  end:              Mapped[Optional[datetime]] = mapped_column(
   server_onupdate=func.now())
  status:           Mapped[str]
  flashencryption:  Mapped[bool] = mapped_column(default=False)
  secureboot:       Mapped[bool] = mapped_column(default=False)
  memoryprotection: Mapped[bool] = mapped_column(default=False)
```

Listing 5.16: SQLAlchemy ORM Models

5.3 Client

The client is the ESP32-S3 and the firmware running on it. On request, it extracts its security configuration and sends it back to the server. In this section, the file structure and selected code snippets of the client project are described. The firmware is developed with the C programming language [45] and the ESP Integrated Development Framework [12].

5.3.1 Overview

The client project contains the source code for the firmware running on the ESP32-S3. This includes the ESP-DHS client and the user application. The following list explains the files and folders of the project, also shown in Figure 5.3.

- The `build` directory is automatically generated when the firmware is built and contains the binaries.
- The `main.c` file inside the `main/` directory is the entry point of the application.
- `CMakeLists.txt` files define dependencies on other components and are necessary for the build system.
- The `components/` directory contains custom components that provide functionality that can be used throughout the code.
- The `dhs_mqtt/` component is responsible for connecting the ESP-DHS client to the MQTT server. The content of the other component directories is very similar.
  - The `dhs_mqtt.h` file inside the `include/` directory contains declarations for component interface functions.
  - The certificate `ca.crt` is specific to this component and is necessary to create a TLS-secured connection between the ESP-DHS client and the MQTT broker.
  - The `dhs_mqtt.c` file contains the logic of the component.
  - The `Kconfig.projbuild` file defines configurations for the component that can be managed by the Kconfig system.
- The `dhs_atecc/` component is responsible for the connection and communication between the ESP32-S3 and the ATECC608B. It provides an interface to the rest of the code to leverage the functionality of the secure element.
- The `dhs_configuration/` component is responsible for extracting the security configuration of the device and encoding it in JSON format.
- The `dhs_wifi/` component connects the device to a Wi-Fi network.
- The `ua_main/` component is the entry point of the user application.
- The `ua_blink/` component contains the logic to let the built-in LED of the ESP32-S3 blink in different colors.
- The `managed_components/` directory is automatically generated and contains components that were automatically downloaded from a component repository.
- The `partitions.csv` file defines what partitions the ESP32-S3 flash memory is divided in.
- The `sdkconfig` file contains the values of the project configuration.
- The `sdkconfig.defaults` file contains default configurations that are set automatically on the first build of the project.
- The `secure_boot_key.pem` is the signing key for secure boot and might be generated by the ESP-DHS server.
- The `README.md` file contains instructions for setting up the project.

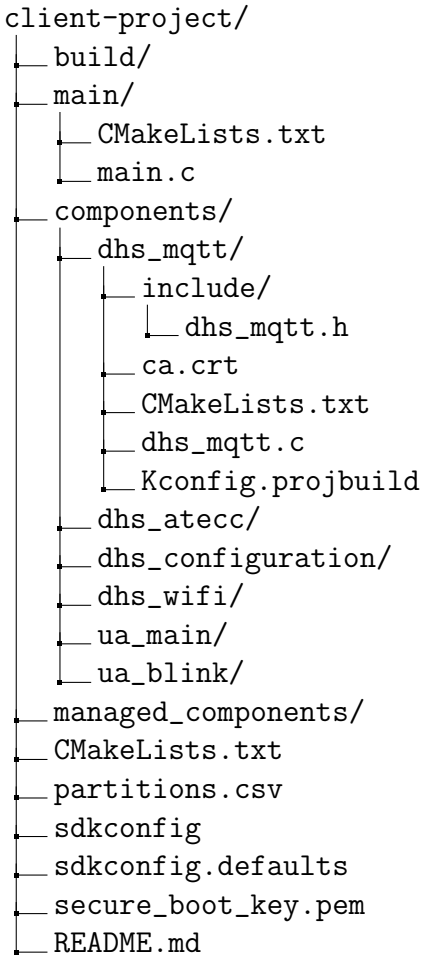


Figure 5.3: File Structure of the Client Project

5.3.2 Project Configuration

The project is configured with the Kconfig mechanism. The configuration can be edited in a GUI by running the command `idf.py menuconfig` from the ESP-IDF terminal, which can be seen in Figure 5.4. Alternatively, the Visual Studio Code extension can be used. The following sections provide an overview of the most important project configuration options.

DHS ATECC608 Configuration

This menu belongs to the `dhs_atecc` component, `ATECC_CONNECTED` is its only configuration. If this option is not selected, the configuration module will not check whether a secure element was connected. Since checking whether a secure element takes a relatively long time, the option can speed up the performance of the application if it is known beforehand that no secure element will be used.

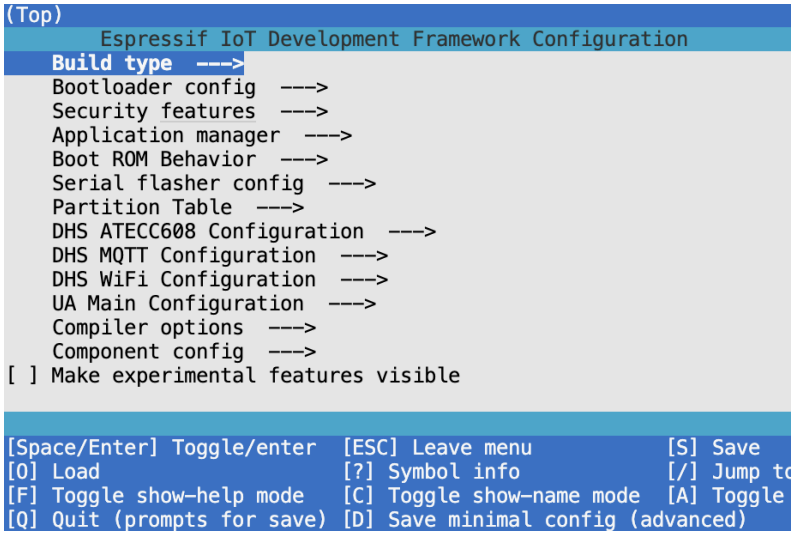


Figure 5.4: UI of the Client Project Configuration

DHS MQTT Configuration

The `BROKER_URI` configuration of the MQTT menu is used to provide the connection URL of the broker. The URL is of format `mqtts://<username>:<password>@<URL>:<port>`.

DHS Wi-Fi Configuration

In the Wi-Fi menu, the Wi-Fi SSID (`ESP_WIFI_SSID`) and password (`ESP_WIFI_PASSWORD`) are provided. It is also possible to define the maximal number of retries that should be performed after an unsuccessful connection attempt (`ESP_MAXIMUM_RETRY`).

Debugging Configurations

To ensure flash encryption and secure boot can be properly debugged, it is necessary to set some configurations that prevent actual eFuses from being burned. For that, the eFuses are simulated in RAM by activating `EFUSE_VIRTUAL` and stored in flash memory by `EFUSE_VIRTUAL_KEEP_IN_FLASH` to be available again after a device restart. This requires an additional flash memory partition. The `CONFIG_PARTITION_TABLE_CUSTOM` option needs to be checked and the path to the `partitions.csv` file that defines the custom partition table needs to be assigned to `CONFIG_PARTITION_TABLE_CUSTOM_FILENAME`.

UA Main Configuration

UA Main is the user application configuration menu. `UA_DURATION` specifies in milliseconds how long the LED blinks in one color. If `UA_LED_ACTIVE` is deactivated, the LED is turned off and `UA_LOGS_ACTIVE` can be used to disable log messages to the console by the user application.

5.3.3 Application Entry Point

The `app_main()` function in the `main` component shown in Listing 5.17 is the entry point of the firmware. First, the secure element is initialized if the configuration indicates that it should be connected. Then, the security configuration is extracted and printed to the console for debugging purposes. To run the ESP-DHS, NVS, Wi-Fi, and MQTT are initialized. At the end, the user application is started.

```
// main.c

void app_main(void)
{
  if(ATECC_CONNECTED) {
    dhs_atecc_init();
  }

  char* config = dhs_config_get();
  printf("%s\n", config);
  free(config);

  // initialize ESP-DHS
  ESP_ERROR_CHECK(nvs_flash_init());
  dhs_wifi_init();
  dhs_mqtt_init();

  ua_main_init();
}
```

Listing 5.17: Example JSON of the Security Configuration

5.3.4 Security Configuration

The security configuration is extracted in the `dhs_configuration` component as shown in Listing 5.18. There exist different ways to extract the state of security features. For MAC addresses, chip info, flash encryption, and secure boot, the ESP-IDF provides dedicated functions to read out the state. The secure element status check is provided by the `dhs_atecc` module, the download mode disabled bit needs to be read out directly from eFuses.

The cJSON library [46] is used to create the JSON string. The `*configuration` pointer references a cJSON object, to which key-value pairs are appended. For example, properties of type string are appended using `cJSON_AddStringToObject()`. The object is converted to a string with `cJSON_Print()` and then deleted with `cJSON_Delete()` to free the memory. To avoid memory leaks, `*json_res` also needs to be freed after usage.

```c
// dhs_configuration.c

#include <cJSON.h>

char* dhs_config_get()
{
  cJSON *configuration = cJSON_CreateObject();

  char buffer[18];

  uint8_t base_mac_address[6];
  esp_base_mac_addr_get(base_mac_address);
  format_mac_address(base_mac_address, buffer);
  cJSON_AddStringToObject(configuration, "base_mac_address", buffer);

  const char *idf_version = esp_get_idf_version();
  cJSON_AddStringToObject(configuration, "esp_idf_version",idf_version);

  esp_chip_info_t chip_info;
  esp_chip_info(&chip_info);

  cJSON_AddNumberToObject(configuration, "chip_cores", chip_info.cores);

  bool flash_encryption_enabled = esp_flash_encryption_enabled();
  cJSON_AddBoolToObject(configuration, "flash_encryption_enabled",
   flash_encryption_enabled);

  bool secure_boot_enabled = esp_secure_boot_enabled();
  cJSON_AddBoolToObject(configuration, "secure_boot_enabled",
   secure_boot_enabled);

  bool memory_protection_enabled = MEMORY_PROTECTION_ENABLED;
  cJSON_AddBoolToObject(configuration, "memory_protection_enabled",
   memory_protection_enabled);

  bool download_mode_disabled = esp_efuse_read_field_bit(
   ESP_EFUSE_DIS_DOWNLOAD_MODE);
  cJSON_AddBoolToObject(configuration, "download_mode_disabled",
   download_mode_disabled);

  bool atecc_connected = false;
  if(ATECC_CONNECTED) { // config indicates ATECC should be connected
    // check if ATECC is connected
    atecc_connected = dhs_atecc_get_status();
  }
  cJSON_AddBoolToObject(configuration, "atecc_connected",
   atecc_connected);

  char *json_res = cJSON_Print(configuration);
  cJSON_Delete(configuration);
  return json_res;
}
```

Listing 5.18: Excerpt of the Code for Extracting the Security Configuration

The structure of the JSON containing the security configuration is shown in Listing 5.19. The properties are explained in the following list:

- `base_mac_address` is the manually set address.
- `efuse_default_mac` is the by Espressif factory programmed MAC address.
- `efuse_custom_mac` is the MAC address written to the eFuse Block 3.
- `esp_idf_version` is the IDF version used to develop the firmware.
- `chip_cores` is the number of processor cores of the microcontroller.
- `chip_features` is the bit mask of feature flags.
- `chip_model` is the type of chip, e.g. ESP32-S3.
- `chip_revision` is the chip revision number.
- `flash_encryption_enabled` tells whether flash encryption is activated.
- `flash_encryption_mode` can either be disabled (`0`), development (`1`) or release (`2`).
- `secure_boot_enabled` tells whether secure boot is activated.
- `aggressive_key_revoke_enabled` tells whether keys are revoked aggressively.
- `memory_protection_enabled` tells whether memory protection is activated.
- `memory_protection_locked` tells whether the memory protection setting can be changed.
- `download_mode_disabled` tells whether new firmware can be flashed onto the device.
- `security_download_enabled` tells whether arbitrary code can be downloaded through the UART port.
- `anti_rollback_secure_version` is the latest security version of an OTA flashed firmware. Firmware with lower secure versions cannot be flashed.
- `atecc_connected` tells whether the ATECC608B is connected. If the value is `false`, it may either be that the secure element is not intended to be used (which must be defined in the project configuration) or that it got disconnected.

```
{
  "base_mac_address": "F4:12:FA:E3:47:5C",
  "efuse_default_mac": "F4:12:FA:E3:47:5C",
  "efuse_custom_mac": "0:0:0:0:0:0",
  "esp_idf_version": "v5.2.1-dirty",
  "chip_cores": 2,
  "chip_features": 18,
  "chip_model": 9,
  "chip_revision": 1,
  "flash_encryption_enabled": false,
  "flash_encryption_mode": 0,
  "secure_boot_enabled": false,
  "aggressive_key_revoke_enabled": false,
  "memory_protection_enabled": false,
  "memory_protection_locked": false,
  "download_mode_disabled": false,
  "security_download_enabled": false,
  "anti_rollback_secure_version": 0,
  "atecc_connected": true
}
```

Listing 5.19: Example JSON of the Security Configuration

5.3.5 MQTT Messaging

A prerequisite for MQTT messaging is a working Wi-Fi connection. The code of the Wi-Fi component was taken from the Wi-Fi Station Example [47] of the ESP-IDF examples repository and modified for the use case. The code for the MQTT component is based on the MQTT SSL Example [48] and was extended with custom logic.

Listing 5.20 shows the MQTT event handler, which runs in the client event loop and permanently listens for dispatched events, which are explained in the following list:

- `MQTT_EVENT_CONNECTED` is dispatched when the connection with the broker is established. In this case, the client subscribes to the `/config-request` topic and then publishes its security configuration on the `/device-connected` topic.
- `MQTT_EVENT_DISCONNECTED` is dispatched when the client disconnects from the MQTT broker. The event is logged to the console.
- `MQTT_EVENT_SUBSCRIBED` is dispatched when the subscription to a topic is successful.
- `MQTT_EVENT_PUBLISHED` is dispatched when a message is published successfully.
- `MQTT_EVENT_DATA` is dispatched when a message is received. If it happened on the `/config-response` topic, the configuration JSON is extracted and appended to the payload of a message that is published to the `/config-response` topic.
- `MQTT_EVENT_ERROR` is dispatched when an error occurs in the event loop. Errors are logged to the console.

```
// dhs_mqtt.c

static void mqtt_event_handler(void *handler_args, esp_event_base_t base
    , int32_t event_id, void *event_data)
{
  esp_mqtt_event_handle_t event = event_data;
  esp_mqtt_client_handle_t client = event->client;
  int msg_id;
  switch ((esp_mqtt_event_id_t)event_id) {
    case MQTT_EVENT_CONNECTED:
      ESP_LOGI(TAG, "MQTT_EVENT_CONNECTED");
      msg_id = esp_mqtt_client_subscribe(client, "/config-request", 0);
      ESP_LOGI(TAG, "sent subscribe successful, msg_id=%d", msg_id);
      char *data = dhs_config_get();
      msg_id = esp_mqtt_client_publish(client, "/device-connected", data
    , 0, 2, 0);
      free(data);
      break;
    case MQTT_EVENT_DISCONNECTED:
      ESP_LOGI(TAG, "MQTT_EVENT_DISCONNECTED");
      break;
    case MQTT_EVENT_SUBSCRIBED:
      ESP_LOGI(TAG, "MQTT_EVENT_SUBSCRIBED, msg_id=%d", event->msg_id);
      break;
    case MQTT_EVENT_UNSUBSCRIBED:
      ESP_LOGI(TAG, "MQTT_EVENT_UNSUBSCRIBED, msg_id=%d", event->msg_id)
    ;
      break;
    case MQTT_EVENT_PUBLISHED:
      ESP_LOGI(TAG, "MQTT_EVENT_PUBLISHED, msg_id=%d", event->msg_id);
      break;
    case MQTT_EVENT_DATA:
      ESP_LOGI(TAG, "MQTT_EVENT_DATA");
      if(strcmp(event->topic, "config-request")) {
          char *data = dhs_config_get();
          msg_id = esp_mqtt_client_publish(client, "/config-response",
    data, 0, 2, 0);
          free(data);
      }
      break;
    case MQTT_EVENT_ERROR:
      // log errors to the console
      break;
    default:
      ESP_LOGI(TAG, "Other event id:%d", event->event_id);
      break;
    }
}
```

Listing 5.20: Event Handler for MQTT Messages

5.3.6 ATECC608B Connection

The ATECC608B component is based on the Secure Element Example [49] and was extended with custom functionality. Listing 5.21 shows the code responsible for checking whether the secure element is connected. The `dhs_atecc_get_status()` function calls `atca_ecdsa_test()`, which performs an Elliptic Curve Digital Signature Algorithm (ECDSA) test by letting the ATECC608B first create and then verify an ECDSA signature. If the return code is `0`, the test was successful, the secure element works as expected and `dhs_atecc_get_status()` returns `true`. For any other return code, `false` is returned.

```
// dhs_atecc.c

#include "cryptoauthlib.h"
#include "mbedtls/atca_mbedtls_wrap.h"

static int atca_ecdsa_test()
{
    mbedtls_pk_context pkey;
    int ret;
    unsigned char buf[MBEDTLS_MPI_MAX_SIZE];
    size_t olen = 0;

    // generate ECDSA signature
    ret = atca_mbedtls_pk_init(&pkey, 0);
    if (ret != 0) goto exit;

    ret = mbedtls_pk_sign(&pkey, MBEDTLS_MD_SHA256, hash, 0, buf,
    MBEDTLS_MPI_MAX_SIZE, &olen, mbedtls_ctr_drbg_random, &ctr_drbg);
    if (ret != 0) goto exit;

    // verifying ECDSA signature
    ret = mbedtls_pk_verify(&pkey, MBEDTLS_MD_SHA256, hash, 0, buf, olen
    );
    if (ret != 0) goto exit;
    mbedtls_pk_free(&pkey);

exit:
    mbedtls_pk_free(&pkey);
    fflush(stdout);
    return ret;
}

bool dhs_atecc_get_status() {
    int ret = atca_ecdsa_test();
    if (ret != 0) {
        ESP_LOGE(TAG, "ECDSA sign/verify failed");
        return false;
    }
    return true;
}
```

Listing 5.21: Checking the Secure Element Connection

In the `dhs_atecc` module, custom functionality can be defined to provide secure element capabilities to the user application. Listing 5.22 shows how the `atcab_random()` function can be leveraged to get a random number from the ATECC608B. This functionality can be accessed in other parts of the code via the `dhs_atecc_get_random_number()` function.

```
#include "cryptoauthlib.h"
#include "mbedtls/atca_mbedtls_wrap.h"

bool dhs_atecc_get_random_number(uint8_t buf[32]) {
    int ret = atcab_random(buf);
    if (ret != 0) {
        ESP_LOGE(TAG, "Failed to get random number: %d", ret);
        return false;
    }
    ESP_LOGI(TAG, "Random number successfully obtained");
    return true;
}
```

Listing 5.22: Random Number Generation using ATECC608B

5.3.7 User Application

The user application performs the actual functionality for the intended microcontroller use case. As an example, a user application that blinks the built-in LED of the ESP32-S3 in random colors is implemented. The code of the `ua_blink` module is based on the Blink Example [50] of the ESP-IDF example repository.

Listing 5.23 shows the implementation of the user application in the `ua_main` module. Random numbers for the application are generated by the ATECC608B and retrieved by the `dhs_atecc_get_random_number()` function. The random number is an array of 32 8-bit integers. Each of these integers has a range from 0 to 255. The random colors are generated by looping through the integer array and using the values of three consecutive integers as activation values for the RGB channels of the LED.

```
void ua_main_init()
{
  if(!ATECC_CONNECTED) {
    ESP_LOGI(TAG, "User application cannot run without connected
   ATECC608.");
    return;
  }
  ua_blink_init();
  uint8_t random_number[32];
  int ret = dhs_atecc_get_random_number(random_number);
  if(ret == 0) {
    ESP_LOGE(TAG, "Cannot get random number.");
  } else {
    while(true) {
      for(int i=0; i<27; i+=3) {
        uint8_t red = random_number[i];
        uint8_t green = random_number[i+1];
```

```
17          uint8_t blue = random_number[i+2];
18          if(LOGS_ACTIVATED) {
19            ESP_LOGI(TAG, "Current color (r,g,b): %d,%d,%d", red, green,
    blue);
20          }
21          if(LED_ACTIVATED) {
22            ua_set_led_color(red, green, blue);
23          }
24          vTaskDelay(CONFIG_UA_DURATION / portTICK_PERIOD_MS);
25        }
26      }
27    }
28  }
```

Listing 5.23: User Application Main Loop

5.4 MQTT Broker

The broker is the gateway through which all MQTT messages between the ESP-DHS server and the client are routed. Mosquitto [51], which is an open-source message broker, was used for the ESP-DHS. The broker was installed on the same machine as the ESP-DHS server, but could theoretically reside on any machine. To ensure only authorized entities can connect to the broker and messages cannot be read by unauthorized parties, the broker is secured with TLS as well as username and password authentication. A guide on how to set up the Mosquitto broker is in Appendix A.

Mosquitto provides the `mosquitto_passwd` command line utility to create password files. Listing 5.24 shows an example of such a file, usernames are followed by a colon and the encrypted password.

```
1 # passwords.txt
2
3 user1:$7$101$sFoRmj4caqK...Bxi3ypw==
4 user2:$7$578$6PXuttS89qv...TFmtuIa==
```

Listing 5.24: Mosquitto Password File

The broker can be configured in the `mosquitto.conf` file, an example is shown in 5.25. The password file needs to be referenced in that file. The listener option specifies on which port the broker listens for incoming messages, the standard value for TLS-secured brokers is port 8883.

To set up TLS encryption, the on most devices pre-installed `openssl` command line tool can be used to create the necessary certificates. Three files need to be generated:

- `ca.crt` is the certificate authority certificate. This file is needed on the broker and on all clients that want to connect to it. In the mosquitto.conf file, it is referenced by the `cafile` option.

- `server.key` holds the private key and is only stored on the broker. It is referenced by the `keyfile` option.
- `server.crt` is the server certificate and is only stored on the broker. It is referenced by the `certfile` option.

```
# mosquitto.conf

listener 8883
password_file /user/michel/mosquitto/passwords.txt
cafile /user/michel/mosquitto/ca.crt
keyfile /user/michel/mosquitto/server.key
certfile /user/michel/mosquitto/server.crt
```

Listing 5.25: Mosquitto Configuration File

5.5 Summary

This chapter provided the most relevant implementation details from the different ESP-DHS components. The following paragraphs summarize the implementation of each component.

The **server** is programmed in Python [41] and configured with a `.env` file that contains environment variables like the path to the database file, broker credentials, or request interval. MQTT messaging is implemented with the Paho library [42]. The main loop continuously publishes messages to the `/config-request` topic and listens on the `/config-response` topic for the security configuration of the ESP32-S3 and on the `/device-connected` topic whether the client got connected to the broker. If received, it is checked whether secure boot, flash encryption, and memory protection are enabled. If not, the client project configuration is changed with the Kconfiglib [25] library. For compiling and flashing the firmware with updated security settings, the ESP-IDF command line tool [12] and esptool [43] are used. For safe debugging, eFuses are simulated in memory, which needs to be disabled when using ESP-DHS in production. All messages, configurations, and flashes are stored in the database using the SQLAlchemy library [36].

For the **database**, SQLite 3 [44] was used. The database schema has three tables: messages, configurations, and flashes. Since, every configuration was received by a message but not every message contains a configuration, the messages and configurations tables have an optional one-to-one relationship. The flashes table is independent of the messages. Tables are represented in Python code as SQLAlchemy ORM models.

The **client** project contains the firmware running on the ESP32-S3 which consists of the ESP-DHS client and the user application. It is developed with the C programming language [45] and ESP-IDF [12]. The client project is configured with the Kconfig mechanism [23] and contains standard options like the secure boot activation but also custom options like the MQTT server path. The security configuration can be extracted by various mechanisms like built-in functions from the ESP-IDF or reading options directly from eFuses.

Converting the configuration to a JSON string is done with the cJSON library [46]. The components for connecting to the Wi-Fi, MQTT broker, and ATECC608B are based on examples provided by the ESP-IDF. If the connection to the broker was successful, the client publishes a message to the `/device-connected` topic. It also subscribes to the `/config-request` topic and sends the configuration to the `/config-response` topic upon request. The user application lets the built-in LED of the ESP32-S3 blink in different colors. This is done by getting an array of random numbers from the ATECC608B and looping through it to get the values for the red, green, and blue channels of the LED.

The **MQTT broker** distributes the messages between client and server is set up with the open-source Mosquitto software [51]. It is secure with a username and password and is set up to require a TLS secure connection with all its clients.

Chapter 6

Evaluation

In this chapter, the resource consumption, performance, and security enhancements of the ESP-DHS are discussed. Furthermore, it is shown how the ESP-DHS provides a potential new attack surface and what has been done to minimize it.

6.1 Resource Consumption

6.1.1 Memory Usage

Memory usage of the ESP32-S3 can be analyzed with the `idf.py size` command of the ESP-IDF. The used microcontroller has two megabytes of non-volatile flash memory, which is used to store the firmware binary. Table 6.1 shows the usage of the different memory types of the ESP-DHS without user application, Table 6.2 shows the memory usage with user application.

Flash memory is the non-volatile memory in which the firmware binary is stored. DRAM is used to hold data, while IRAM is used for instructions. D/IRAM can be used for both data and instructions. Depending on the use case, it can vary how much memory is allocated to IRAM and D/IRAM, but the combined value is always roughly 708 KB.

The ESP-DHS without user application uses 818.4 KB (40.9%) of flash memory, 86.4 KB (23.8%) of static IRAM and 35.3 KB (10.2%) of static D/IRAM. The example user application only uses an additional 19 KB ($\sim$ 1%) of flash memory, static D/IRAM usage stays the same. For simple IoT applications, this might be enough if the user application can leverage functionality from the ESP-DHS, like the secure element interface or Wi-Fi connection. However, for more complex functionality, it is recommended to use an ESP32 model with more memory or to connect external RAM [52].

Memory Type	Available	Used (Absolute)	Used (%)
Flash Memory	2000 KB	818.4 KB	40.9%
Static IRAM	362.2 KB	86.4 KB	23.8%
Static D/IRAM	345.9 KB	35.3 KB	10.2%

Table 6.1: ESP-DHS Memory Usage without User Application

Memory Type	Available	Used (Absolute)	Used (%)
Flash Memory	2000 KB	837.5 KB	41.9%
Static IRAM	362.2 KB	91.2 KB	25.2%
Static D/IRAM	345.9 KB	35.4 KB	10.2%

Table 6.2: ESP-DHS Memory Usage with User Application

6.1.2 CPU Usage

CPU usage was measured with the `esp32-perfmon` component [53], which is not a standard part of the ESP-IDF. When the CPU cores are idle, they run specified idle functions, which increase an idle counter. The counter is divided by the expected number of the counter if the CPU were idle all the time. The resulting value is subtracted from one to get the CPU usage in percent. The usage is calculated for five-second intervals, e.g. a value of 10% means that the CPU was used for 10% of the time in the last five seconds.

CPU usage of the ESP-DHS is summarized in Table 6.3. The ESP-DHS is idle when no MQTT messages are handled and the security configuration is not being extracted. Only the MQTT server uses the CPU to listen passively for incoming requests. In this state, CPU usage for core 0 is only 3% and 2% for core 1. In the interval where an incoming configuration request is handled and the security configuration is extracted, core 0 usage increases to 7%, and core 1 decreases to 1%. If the ATECC608B is not connected, although it should be, core 0 usage is 4% and core 1 usage is 2% in the interval where the security configuration is returned. The CPU usage is overall very low and leaves enough room for computationally heavy user applications.

Interval	Core 0	Core 1
Idle	3%	2%
Request Processing and Configuration Extraction	7%	1%
Configuration Extraction (ATECC608B disconnected)	4%	2%

Table 6.3: ESP-DHS CPU Usage

6.2 Performance

6.2.1 Configuration Extraction

The duration for extracting the security configuration was measured by calling the function `esp_timer_get_time()` before and after the call of `dhs_config_get()` and taking the difference of the returned times. The extraction takes around 216 milliseconds. However, if the secure element is not properly connected, this time drastically increases to over 44 seconds. This slows down the whole ESP-DHS since the function call blocks other parts of the program. If it is known beforehand that no secure element will be connected, the `ATECC_CONNECTED` option in the project configuration must be disabled, which will skip the time costly `dhs_atecc_get_status()` function and always returns `false` for the `atecc_connected` property of the security configuration JSON, as shown in Listing 5.18.

6.2.2 Building and Flashing Firmware

The speed of building and flashing depends largely on the computational power of the host machine on which the ESP-DHS server runs. The host machine used for the tests is a MacBook Air M2 from 2022. The measurements were made by calling `datetime.now()` before and after the call of `compile_secure()` function and taking the difference of the returned times.

Building and flashing the client project takes 1 minute and 27 seconds if all features need to be activated. Activating only one feature is up to 3 seconds faster. Table 6.4 shows the time taken depending on which security features were activated. The measurements deviated in different tests only by a few milliseconds.

Activated Feature	Build and Flash Time
Secure Boot	1 min 24 s
Flash Encryption	1 min 26 s
Memory Protection	1 min 24 s
All Features	1 min 27 s

Table 6.4: Build and Flash Time for Security Features

6.2.3 Workflow

Workflow performance is measured by analyzing the timestamps stored in the database. There are two workflow scenarios to consider, depending on whether the firmware needs to be updated. If not, the ESP-DHS only has one round trip, by requesting the security configuration and getting the response. An additional round trip is added when the configuration is not secure, by flashing the firmware and getting the "device connected" response.

Step	Execution Time	Difference
Config Request Sent	0	0
Config Response Received	0.29 s	+ 0.29 s
Flash Start	0.30 s	+ 0.01 s
Flash End	1 min 26.12 s	+ 1 min 25.83 s
Device Connected Received	1 min 52.53 s	+ 26.41 s

Table 6.5: Workflow Measurements

The round trip time is influenced by the computational power of the host computer and the microcontroller and the network speed. Table 6.5 shows how much time the steps need to execute and the time difference to the last step. The configuration request is sent at time 0. After 0.29 seconds, the response with the ESP32-S3 security configuration is received. After another 0.01 seconds, compiling and flashing the firmware begins, which takes 1 minute and 25.83 seconds. The microcontroller sends the "device connected" message after it has rebooted, which takes 26.41 seconds. The overall workflow from sending the configuration request until the ESP32-S3 has booted the new secure firmware takes 1 minute and 52.53 seconds in the test environment. The measurements were made multiple times and revealed only small differences by a few milliseconds between the different test runs.

6.3 Security Considerations

The ESP-DHS hardens the ESP32-S3 by activating secure boot, flash encryption, and memory protection. Furthermore, it provides support for the ATECC608B secure element. The security benefits of these features are discussed on a high level, conducting attacks on the hardened microcontroller to test the measure's effectiveness is beyond the scope of this thesis.

6.3.1 Secure Element

The effectiveness of using a secure element for cryptographic operations and key storage was shown in [20] and summarized in Section 3.2. The authors used an ESP32 microcontroller with the ATECC608A, which are very similar devices to the ones used in this thesis. Therefore the results should be transferrable to a big extent. It could be shown that the ESP32 is vulnerable to format string attacks, a type of attack where more memory segments than intended are read out when printing to the console. If this happens to memory segments containing cryptographic keys, this poses a security risk. By using a secure element, this can be avoided.

In the implementation of this thesis, the ATECC608B was connected to the ESP32-S3 via GPIO pins. If an attacker has physical access to the device, he could disconnect the secure element from the microcontroller and potentially break the application running on it. An attacker could also intercept the traffic between the microcontroller and the secure

element. To mitigate these security risks in a production environment, it is advisable to use a microcontroller where the secure element is on the same circuit board as the CPU or inside the chip.

6.3.2 Secure Boot

Secure boot ensures that only signed firmware can run on the ESP32-S3 microcontroller [18]. Since the signing key created by the ESP-DHS server only exists on the host computer, this is an effective countermeasure to prevent malicious code from being run on the microcontroller. It is crucial that this key is kept confidential, this is most effectively done by restricting access to the host computer and the directory where the secure boot signing key is stored.

6.3.3 Flash Encryption

Flash encryption is used to encrypt the flash memory of the ESP32-S3, which ensures that an attacker cannot read out data and firmware from the device. The flash encryption key is generated on the microcontroller itself and stored in eFuses [19]. The ESP-DHS activates flash encryption only in development mode to allow for debugging. To ensure the highest level of security, production mode should be activated.

6.3.4 Memory Protection

Memory protection prevents the program running on the ESP32-S3 from reading and writing memory regions without explicit permission. Permissions are granted automatically by the microcontroller [16]. To exploit disabled memory protection, malicious firmware would need to run on the microcontroller, which is prevented by secure boot.

6.3.5 ESP-DHS Attack Surface

While the ESP-DHS hardens the ESP32-S3, it also introduces an additional layer of complexity and therefore a new attack surface. Minimizing this attack surface could be part of future work and is discussed in Chapter 7.

The security configuration contains critical information about potential device vulnerabilities and must be kept confidential. It is exchanged over the MQTT protocol, the ESP-DHS makes sure that this connection is TLS secured and messages cannot be read by unauthorized parties in a man-in-the-middle attack. The security configuration is stored in an SQLite database, which does not provide encryption or password protection by default. It is therefore crucial that no unauthorized parties have access to the host computer on which the database file is stored.

New firmware is flashed over a physical UART connection between the host computer and the ESP32-S3, which is not secured. If secure boot is not activated, the UART connection is susceptible to man-in-the-middle attacks, where an attacker could read the firmware binary. In the flashing process, after secure boot was first activated, the secure boot key is transferred from the host computer to the microcontroller over this connection. If an attacker has access to the connection in this stage of the process, he could decrypt the firmware binaries and sign malicious firmware.

Another security risk is that the ESP-DHS client is not isolated from the rest of the firmware running on the ESP32-S3. On-device malware could intercept and forge the security configuration before it is sent to the server.

The ESP-DHS cannot make sure that the device sending the security configuration over MQTT and the device receiving the firmware over UART are the same. This could lead to a scenario where one ESP-DHS client sends a device configuration where all security features are activated, but the ESP32-S3 connected to the UART port has a different configuration. The device connected to the port could then be used in production without activated security features.

6.4 Summary

The ESP-DHS uses a big part of the ESP32-S3 flash memory, while the usage of static IRAM and D/IRAM is moderate. 40.9% (818.4 KB) of the 2 MB flash memory of the used microcontroller gets occupied by the ESP-DHS. Static IRAM usage is 23.8% (86.4 KB), D/IRAM usage is 10.2% (35.3 KB). When developing a user application, it is important to make sure to also consider the memory usage of the ESP-DHS. The ESP-DHS is computationally very light, CPU performance was measured by calculating CPU occupation in a five-second interval. Only 3% of the CPU core 0 is used in an idle interval, 7% when the security configuration is extracted, and 4% when the configuration is extracted but the ATECC608B was disconnected. Core 1 usage is always around 2%. This leaves enough computational power for computationally intensive user applications.

The configuration extraction is very fast with 216 milliseconds. However, when the ATECC608B is not connected although it should be, this duration increases to 44 seconds. It is therefore very important to specify in the client project configuration whether a secure element is used or not. Building and flashing firmware with all activated security features takes around 1 minute and 27 seconds on the used host computer (MacBook Air M2 2022), with only very little performance improvement when just a single feature is activated. The workflow of the ESP-DHS has two round trips when the security configuration needs to be updated. In the test setup, a configuration response is received 0.29 seconds after the request. Configuration evaluation, updating the configuration, compiling, and flashing the firmware takes around 1 minute and 25 seconds. The server receives the “device connected” message 1 minute and 52 seconds after it has sent a configuration request. Compiling and flashing the firmware and restarting the microcontroller are the steps that take the longest.

The ESP-DHS hardens the ESP32-S3 by automatically activating secure boot, flash encryption, and memory protection if necessary. Secure boot ensures that only signed firmware can run on the microcontroller, flash encryption prevents attackers from reading out the flash memory contents and memory protection ensures only authorized entities can access the memory of the device. The ESP-DHS also provides an API for accessing the ATECC608B secure element. The usage of a secure element helps to avoid string format attacks that can leak cryptographic keys, which was shown in [20].

The ESP-DHS stores the security configuration in an unencrypted database, the UART connection between the host computer and ESP32-S3 is also not encrypted. It is therefore important to restrict access to the machine hosting the database and to ensure only authorized entities have access to the physical UART connection during the flashing process. The ESP-DHS client does not run isolated from the rest of the firmware, which is a security risk. The ESP-DHS client does not run isolated from the rest of the firmware, therefore, the security configuration could potentially be forged by an on-device malware before it is sent to the server. The system has no way to check whether the device connected to the host computer is the same from which it receives the MQTT messages.

Chapter 7

Conclusion

Section 7.1 of this chapter summarizes the findings of this thesis. The limitations of the implemented solution, how they can be addressed and future research opportunities are discussed in Section 7.2.

7.1 Summary

The main contribution of this thesis was the development of the **ESP**32-S3 **D**evice **H**ardening **S**ystem (**ESP-DHS**), which extracts the security configuration of an ESP32-S3 microcontroller and updates it if necessary. An analysis of related work emphasized the importance of proper device configuration and device hardening to IoT security. Some papers provide concrete measures or frameworks on how to secure IoT devices against attackers. Current solutions for device monitoring focus on monitoring the network traffic and only react if the device is already behaving maliciously. There exist no systems for the ESP32-S3 microcontroller that take a proactive approach by monitoring its security configuration and automatically updating it if necessary. The ESP-DHS aims to close this gap.

The ESP-DHS consists of five parts: a server, database, ESP32-S3 firmware, MQTT broker and ATECC608B secure element. The Python server requests the security configuration of the ESP32-S3 over MQTT. The firmware on the microcontroller extracts the configuration and sends it back to the server, where it is analyzed. The server has access to the firmware source code. If necessary, secure boot, flash encryption, or memory protection are activated in the firmware source code. The new firmware with activated security features is then built and flashed onto the ESP32-S3 over a physical UART connection. All actions taken by the server and all communication between the server and microcontroller are logged to a SQLite database. For additional device security, the firmware has support for the ATECC608B secure element, which provides secure cryptographic functions and key storage. To run custom code next to the ESP-DHS on the microcontroller, it is possible to develop user applications that are integrated into the ESP-DHS firmware.

Evaluation of the ESP-DHS showed that at most 7% of CPU resources of the ESP32-S3 are used in a five-second interval. The firmware uses below 25% of the available 0.7 MB

of RAM. 40.9% of the 2 MB flash storage is occupied by the firmware. The process of adjusting the device configuration and building and flashing new firmware takes 1 minute and 27 seconds in the test environment. The ESP-DHS is therefore computationally very light, however, it is necessary to use a microcontroller with sufficient flash memory since the ESP-DHS can take up a considerable amount of it.

7.2 Limitations and Future Work

While the ESP-DHS can enhance the security of the ESP32-S3, it also introduces a new potential attack surface. The data going through the physical UART connection between the host computer and the ESP32-S3 is not encrypted. The SQLite database also does not offer any encryption and access control capabilities. Another limitation is that the ESP-DHS cannot be sure the device from which it receives the security configuration is the same device connected to the UART port. This could result in the connected device not being flashed with secure firmware. Also, the ESP-DHS client does not run isolated from the user application on the microcontroller. On-device malware could potentially intercept MQTT messages and for example, send a forged security configuration.

Immediate future work building upon the ESP-DHS should first focus on improving these limitations. For example, using MySQL [54] instead of SQLite would provide the database with access rights management and encryption capabilities. For the server to ensure the UART device is the same as the MQTT device, the microcontroller could send its MAC address to the server over UART with the serial protocol. The MAC address received over this channel would be compared to the one received over MQTT to ensure they are the same. To avoid that on-device malware can change the data sent by the ESP32-S3, the ESP-DHS client should be isolated from the rest of the firmware.

The practicability of the ESP-DHS in real-world use cases is limited because the ESP32-S3 needs to be physically connected to the host computer. The system continues working when the device is not connected, however, it will only request and log the device configuration without being able to change it. The ESP-IDF provides methods to update firmware over the air (OTA) [55] which could be integrated into the system in the future. Flashing firmware with the OTA mechanism provides improved security compared to UART updates and would remove the vulnerable connection between the host computer and the microcontroller. Currently, the ESP-DHS only works with one ESP32-S3. By using OTA updates, the system could be extended to support multiple microcontrollers with only one server.

The ESP-DHS was built specifically for the ESP32-S3 microcontroller, adding support for other Espressif devices that use the ESP-IDF would be relatively easy. Adding support for a broader range of IoT platforms could be considered for future work. Additionally, the ESP-DHS could be enhanced with network monitoring capabilities as proposed by [8], [9], and [10] to detect malicious behavior.

Bibliography

[1] O. Garcia-Morchon, S. Kumar, and M. Sethi, "Internet of Things (IoT) Security: State of the Art and Challenges", Internet Engineering Task Force, Tech. Rep. RFC 8576, 2019. [Online]. Available: https://datatracker.ietf.org/doc/rfc8576.

[2] B. Zhao, S. Ji, W.-H. Lee, *et al.*, "A Large-Scale Empirical Study on the Vulnerability of Deployed IoT Devices", *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 3, pp. 1826–1840, May 2022. DOI: 10.1109/TDSC.2020.3037908.

[3] J. Carrillo-Mondéjar, H. Turtiainen, A. Costin, J. L. Martínez, and G. Suarez-Tangil, "HALE-IoT : Hardening Legacy Internet of Things Devices by Retrofitting Defensive Firmware Modifications and Implants", *IEEE Internet of Things Journal*, vol. 10, no. 10, pp. 8371–8394, May 2023. DOI: 10.1109/JIOT.2022.3224649.

[4] A. Echeverría, C. Cevallos, I. Ortiz-Garces, and R. O. Andrade, "Cybersecurity Model Based on Hardening for Secure Internet of Things Implementation", *Applied Sciences*, vol. 11, no. 7, p. 3260, Apr. 2021. DOI: 10.3390/app11073260.

[5] C. Kelly, N. Pitropakis, S. McKeown, and C. Lambrinoudakis, "Testing And Hardening IoT Devices Against the Mirai Botnet", in *2020 International Conference on Cyber Security and Protection of Digital Service*, Jun. 2020, pp. 1–8. DOI: 10.1109/CyberSecurity49315.2020.9138887.

[6] *IoT device cybersecurity capability core baseline*, NIST IR 8259A, May 2020. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8259A.pdf.

[7] *Security and Privacy Controls for Information Systems and Organizations*, NIST-SP-800-53, Sep. 2020. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r5.pdf.

[8] J. Wang, S. Hao, R. Wen, *et al.*, "IoT-Praetor: Undesired Behaviors Detection for IoT Devices", *IEEE Internet of Things Journal*, vol. 8, no. 2, pp. 927–940, Jan. 2021. DOI: 10.1109/JIOT.2020.3010023.

[9] H. Zahan, M. W. Al Azad, I. Ali, and S. Mastorakis, "IoT-AD: A Framework to Detect Anomalies Among Interconnected IoT Devices", *IEEE Internet of Things Journal*, vol. 11, no. 1, pp. 478–489, Jan. 2024. DOI: 10.1109/JIOT.2023.3285714.

[10] W. Zhang, Y. Meng, Y. Liu, X. Zhang, Y. Zhang, and H. Zhu, "HoMonit: Monitoring Smart Home Apps from Encrypted Traffic", in *ACM SIGSAC Conference on Computer and Communications Security*, Oct. 2018, pp. 1074–1088. DOI: 10.1145/3243734.3243820.

[11] *Espressif Announces the Launch of ESP32 Cloud on Chip and Funding by Fosun Group*. [Online]. Available: https://www.espressif.com/en/media_overview/news/20160907-esp32briefing (visited on 03/22/2024).

[12] *Get Started - ESP32-S3 - ESP-IDF Programming Guide latest documentation*. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/get-started/index.html (visited on 03/22/2024).

[13] *Espressif Leads the IoT Chip Market with Over 1 Billion Shipments Worldwide*. [Online]. Available: https://www.espressif.com/en/news/1_Billion_Chip_Sales (visited on 07/16/2024).

[14] *Announcing ESP32-S3 for AIoT Applications*. [Online]. Available: https://www.espressif.com/en/news/ESP32_S3 (visited on 03/23/2024).

[15] *ESP32-S3-DevKitC-1 v1.1 - ESP32-S3 - ESP-IDF Programming Guide latest documentation*. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/hw-reference/esp32s3/user-guide-devkitc-1.html (visited on 03/24/2024).

[16] *Security - ESP32-S3 - ESP-IDF Programming Guide v5.2.2 documentation*. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/v5.2.2/esp32s3/security/security.html (visited on 07/15/2024).

[17] *eFuse Manager - ESP32-S3 - ESP-IDF Programming Guide v5.2.2 documentation*. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/stable/esp32s3/api-reference/system/efuse.html (visited on 07/15/2024).

[18] *Secure Boot V2 - ESP32-S3 - ESP-IDF Programming Guide v5.2.1 documentation*. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/v5.2.1/esp32s3/security/secure-boot-v2.html (visited on 05/08/2024).

[19] *Flash Encryption - ESP32-S3 - ESP-IDF Programming Guide v5.2.1 documentation*. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/stable/esp32s3/security/flash-encryption.html (visited on 05/08/2024).

[20] B. Pearson, C. Zou, Y. Zhang, Z. Ling, and X. Fu, "SIC^2: Securing Microcontroller Based IoT Devices with Low-cost Crypto Coprocessors", in *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*, Dec. 2020, pp. 372–381. DOI: 10.1109/ICPADS51040.2020.00057.

[21] *ATECC608B*. [Online]. Available: https://www.microchip.com/en-us/product/atecc608b (visited on 07/16/2024).

[22] M. Krohn, *What is MQTT? A practical introduction*, Mar. 2022. [Online]. Available: https://www.opc-router.com/what-is-mqtt/ (visited on 08/07/2024).

[23] *Kconfig Language – The Linux Kernel documentation*. [Online]. Available: https://www.kernel.org/doc/html/next/kbuild/kconfig-language.html (visited on 08/08/2024).

[24] *Project Configuration - ESP32-S3 - ESP-IDF Programming Guide v5.2.1 documentation*. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/stable/esp32s3/api-reference/kconfig.html (visited on 05/08/2024).

[25] U. Magnusson, *Ulfalizer/Kconfiglib*. [Online]. Available: https://github.com/ulfalizer/Kconfiglib (visited on 08/08/2024).

[26] *IoT Security Certification for Device Manufacturers | PSA Certified*. [Online]. Available: https://www.psacertified.org/getting-certified/device-manufacturer/ (visited on 05/07/2024).

[27] N. M. Karie, N. M. Sahri, W. Yang, C. Valli, and V. R. Kebande, "A Review of Security Standards and Frameworks for IoT-Based Smart Environments", *IEEE Access*, vol. 9, pp. 121 975–121 995, 2021. DOI: 10.1109/ACCESS.2021.3109886.

[28] *PSA Certified: IoT Security Framework and Certification*. [Online]. Available: https://www.psacertified.org/ (visited on 04/23/2024).

[29] *Information security, cybersecurity and privacy protection — Information security management systems*, ISO/IEC 27001, Oct. 2022. [Online]. Available: https://www.iso.org/standard/27001.

[30] *ESP32-S3 Series (ESP32-S3, ESP32-S3FN8, ESP32-S3R2, ESP32-S3R8, ESP32-S3R8V, ESP32-S3FH4R2) | PSA Certified*. [Online]. Available: https://www.psacertified.org/products/esp32-s3-series-esp32-s3-esp32-s3fn8-esp32-s3r2-esp32-s3r8-esp32-s3r8v-esp32-s3fh4r2/ (visited on 04/21/2024).

[31] Y. Kurii and I. Opirskyy, "Analysis and comparison of the NIST SP 800-53 and ISO/IEC 27001: 2013", in *Cybersecurity providing in information and telecommunication systems*, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:254045769.

[32] V. Kanade, *ISMS Meaning, Its Working, Benefits, and Best Practices - Spiceworks*. [Online]. Available: https://www.spiceworks.com/it-security/vulnerability-management/articles/what-is-isms/ (visited on 05/09/2024).

[33] S. Yoon and J. Kim, "Remote security management server for IoT devices", in *2017 International Conference on Information and Communication Technology Convergence (ICTC)*, Oct. 2017, pp. 1162–1164. DOI: 10.1109/ICTC.2017.8190885.

[34] R. McPherson and J. Irvine, "Using Smartphones to Enable Low-Cost Secure Consumer IoT Devices", *IEEE Access*, vol. 8, pp. 28 607–28 613, 2020. DOI: 10.1109/ACCESS.2020.2968627.

[35] M. Medwed, V. Nikov, J. Renes, T. Schneider, and N. Veshchikov, "Cyber Resilience for Self-Monitoring IoT Devices", in *2021 IEEE International Conference on Cyber Security and Resilience (CSR)*, Jul. 2021, pp. 160–167. DOI: 10.1109/CSR51186.2021.9527995.

[36] *SQLAlchemy*. [Online]. Available: https://www.sqlalchemy.org (visited on 08/08/2024).

[37] *Miscellaneous System APIs - ESP32-S3 - ESP-IDF Programming Guide v5.3 documentation*. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/stable/esp32s3/api-reference/system/misc_system_api.html (visited on 08/25/2024).

[38] *Networking APIs - ESP32-S3 - ESP-IDF Programming Guide v5.2.1 documentation*. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/stable/esp32s3/api-reference/network/index.html (visited on 05/08/2024).

[39] M. Sabbatini, *Atomis14/esp-dhs-server*. [Online]. Available: https://github.com/Atomis14/esp-dhs-server (visited on 08/26/2024).

[40] M. Sabbatini, *Atomis14/esp-dhs-client*. [Online]. Available: https://github.com/Atomis14/esp-dhs-client (visited on 08/26/2024).

[41] *Welcome to Python.org*. [Online]. Available: https://www.python.org/ (visited on 08/29/2024).

[42] *Paho-mqtt 2.1.0*. [Online]. Available: https://pypi.org/project/paho-mqtt/ (visited on 08/07/2024).

[43] *Esptool.py Documentation - ESP32-S3*. [Online]. Available: https://docs.espressif.com/projects/esptool/en/latest/esp32s3/index.html (visited on 08/08/2024).

[44] *SQLite Home Page*. [Online]. Available: https://www.sqlite.org/ (visited on 08/09/2024).

[45] *C docs - get started, tutorials, reference.* [Online]. Available: https://learn.microsoft.com/en-us/cpp/c-language/?view=msvc-170 (visited on 08/29/2024).
[46] D. Gamble, *DaveGamble/cJSON*. [Online]. Available: https://github.com/DaveGamble/cJSON (visited on 08/10/2024).
[47] *Wi-Fi Station Example*. [Online]. Available: https://github.com/espressif/esp-idf/tree/release/v5.2/examples/wifi/getting_started/station (visited on 08/10/2024).
[48] *ESP-MQTT SSL Sample application*. [Online]. Available: https://github.com/espressif/esp-idf/tree/release/v5.2/examples/protocols/mqtt/ssl (visited on 08/10/2024).
[49] *ECDSA sign/verify Example with ESP32-WROOM-32SE*. [Online]. Available: https://github.com/espressif/esp-idf/tree/release/v5.2/examples/peripherals/secure_element (visited on 08/10/2024).
[50] *Blink Example*. [Online]. Available: https://github.com/espressif/esp-idf/tree/release/v5.2/examples/get-started/blink (visited on 08/10/2024).
[51] *Eclipse Mosquitto*. [Online]. Available: https://mosquitto.org/ (visited on 08/09/2024).
[52] *Support for External RAM - ESP32-S3 - ESP-IDF Programming Guide v5.3 documentation*. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/stable/esp32s3/api-guides/external-ram.html (visited on 08/13/2024).
[53] *Carbon225/esp32-perfmon: ESP IDF CPU usage monitor*. [Online]. Available: https://github.com/Carbon225/esp32-perfmon (visited on 08/14/2024).
[54] *MySQL*. [Online]. Available: https://www.mysql.com/de/ (visited on 08/22/2024).
[55] *Over The Air Updates (OTA) - ESP32-S3 - ESP-IDF Programming Guide v5.3 documentation*. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/stable/esp32s3/api-reference/system/ota.html#secure-ota-updates-without-secure-boot (visited on 08/23/2024).

Abbreviations

AI	Artificial Intelligence
AIoT	Artificial Intelligence of Things
API	Application Programming Interface
CPU	Central Processing Unit
DFA	Deterministic Finite Automaton
DUD	Device Usage Description
DUR	Device Usage Rule
ECDSA	Elliptic Curve Digital Signature Algorithm
ESP-DHS	ESP32-S3 Device Hardening System
ESP-IDF	Espressif Integrated Development Framework
FQDN	Fully Qualified Domain Name
GPIO	General Purpose Input/Output
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IDF	Integrated Development Framework
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
IoT	Internet of Things
ISMS	Information Security Management System
JSON	JavaScript Object Notation
LED	Light Emitting Diode
MB	Megabyte
MAC	Media Access Control
MCU	Microcontroller Unit
MITM	Man in the Middle
ML	Machine Learning
MQTT	Message Queuing Telemetry Transport
NIST	National Institute of Standards and Technology
NLP	Natural Language Processing
NVS	Non-volatile Storage
ORM	Object-relational mapping
OTA	Over-the-Air
PSA	Platform Security Architecture
RAM	Random Access Memory
ROM	Read Only Memory

RoT	Root of Trust
SDN	Software Defined Networking
SoC	System on a Chip
SRAM	Static Random Access Memory
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UA	User Application
UI	User Interface
UART	Universal Asynchronous Receiver/Transmitter
VM	Virtual Machine
WPA	Wi-Fi Protected Access

List of Figures

List of Tables

List of Listings

Appendix A

Installation Guidelines

The host computer on which the client firmware was developed and the server, database and broker run was a MacBook Air M2 2022 with macOS Sonoma 14.6.1. The installation steps may vary for other systems.

A.1 Client Setup

To develop, build and flash ESP32-S3 firmware, the ESP-IDF [12] is needed. For the installation, follow the "Get Started" guide from the ESP-IDF programming guide [12]. **The used version of the ESP-IDF VS Code plugin was v1.7.1, newer versions did not work**.

For the ESP-DHS to work, it is necessary that the firmware was built manually at least once (i.e. the `/build` directory should not be empty).

The following configurations need to be set manually before first building the project (not by writing them manually to a `sdkconfig` file, but by entering the values in the menuconfg UI):

```
# the custom table contains a virtual eFuse partition
CONFIG_PARTITION_TABLE_CUSTOM=y

# bigger offset needed for secure boot bootloader
CONFIG_PARTITION_TABLE_OFFSET=0x10000

# SSID of the Wi-Fi network
ESP_WIFI_SSID=<SSID>

# password of the Wi-Fi network
ESP_WIFI_PASSWORD=<PASSWORD>

# URI of the MQTT broker in the following format:
# mqtts://<username>:<password>@<fqdn>:8883
BROKER_URI=<URI>
```

```
16
17 # y if you intend to use a secure element, n otherwise
18 ATECC_CONNECTED=y/n
19
20 # if ATECC608B will be connected, add these additionally:
21 CONFIG_ATCA_MBEDTLS_ECDSA=y
22 CONFIG_ATCA_MBEDTLS_ECDSA_SIGN=y
23 CONFIG_ATCA_MBEDTLS_ECDSA_VERIFY=y
24 CONFIG_ATCA_I2C_SDA_PIN=14  # or any other used pin
25 CONFIG_ATCA_I2C_SCL_PIN=13  # or any other used pin
```

Listing A.1: Server Configuration

A.2 Mosquitto Setup

Mosquitto [51] is the software used to run the MQTT broker. These installation guidelines assume the broker runs on the same machine as the server.

A.2.1 Install Mosquitto

1. Install mosquitto by running: `brew install mosquitto`
2. Configure the broker
   - Create a password file by running:
     `mosquitto_passwd -c <path to password file> <username>`
   - Add the `listener` option to the `mosquitto.conf` file (located at `/opt/homebrew/Cellar/mosquitto/2.0.18/etc/mosquitto`) in order to not only allow connections from the same machine: `listener 1883`
   - Add the `password_file` option to `mosquitto.conf`:
     `password_file <path to the password file>`
   - Grant permission to the user to read/write the password file by running:
     `chmod 0700 <path to password file>`
3. Start the broker by running: `/opt/homebrew/opt/mosquitto/sbin/mosquitto -c /opt/homebrew/etc/mosquitto/mosquitto.conf`

A.2.2 Setup TLS Encryption

1. Create a key pair for the Certificate Authority (CA) by running:
   `openssl genrsa -out ca.key 2048`
2. Create certificate for CA using private key from step 1 by running:
   `openssl req -new -x509 -days 1826 -key ca.key -out ca.crt`

3. Fill in the requested information, remember the fully qualified domain name (FQDN), which could be e.g. the IP address of the broker. The FQDN can be retrieved by running `hostname -f`.

4. Create server key pair that is used by the broker by running: `openssl genrsa -out server.key`

5. Create certificate request using the server key from before (use FQDN from before as the common name) by running:
`openssl req -new -out server.csr -key server.key`

6. Use CA key to verify and sign the server certificate, this creates the server.crt file:
`openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out server.crt -days 360`

7. Copy the following files to a folder under the mosquitto folder: `ca.crt`, `server.crt`, `server.key`

8. Copy `ca.crt` to the root directory of the ESP-DHS server and `/components/dhs_mqtt` directory of the ESP32-S3 firmware project

9. Edit the `mosquitto.conf` file:

   - Change `listener 1883` to `listener 8883`
   - Add `cafile <path to ca.crt>`
   - Add `keyfile <path to server.key>`
   - Add `certfile <path to server.crt>`

A.3 Server Setup

To run the server, a Python installation version 3.9.6 or higher is necessary. The server needs access to the directory containing the source code of the ESP32-S3 firmware.

1. Install the necessary Python packages on your machine:

   - `esptool` for flashing firmware onto the ESP32 over USB
   - `paho-mqtt` for the MQTT connection over TCP
   - `dotenv` to read environment variables from the `.env` file
   - `kconfiglib` to programmatically edit the sdkconfig files
   - `SQLAlchemy` for the communication with the SQLite database

2. Copy the `.env.example` file, rename it to `.env` and adjust the environment variables as needed.

3. To start the server, run the `main.py` file.

Appendix B

Submitted Documents

The following documents were handed in:

1. The midterm presentation of this thesis.
2. The LaTeX source code and PDF.
3. Links to the GitHub repositories [39], [40].
4. The abstract as .txt files in English and German.