UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

Bootstrapping the Integrity of Sensor Data Labels at the Microcontroller Level Using Physically Unclonable Functions

Addressing Physical Vulnerabilities in the IoT Domain

Henrik Monsen

INF-3990 Master's Thesis in Computer Science - May 2024

Supervisors

Main supervisor:	Elisavet Kozyri	UiT The Arctic University of Norway, Faculty of Science and Technology, Department of Computer Science
Co-supervisor:	Tor-Arne Schmidt Nordmo	UiT The Arctic University of Norway, Faculty of Science and Technology, Department of Computer Science

Abstract

Modern decision-making processes across industries today increasingly rely on data-driven insights derived from various sources. As smart devices, sensor technology, and the IoT (Internet of Things) evolve, organizations are progressively leveraging these technologies for data-driven decision-making. However, with the introduction of regulations such as the General Data Protection Regulation (GDPR) in recent years, organizations are compelled to adjust to new limitations imposed on user data collection and processing. This thesis is dedicated to one of the many technical difficulties associated with GDPR compliance in the IoT domain, specifically, compliance with regulations requiring data provenance at the IoT device level.

The thesis investigates the feasibility of leveraging Physically Unclonable Functions (PUFs) to bootstrap the integrity guarantees of sensor data labels, acting as provenance information, especially in environments prone to physical data extraction threats. The work to address the feasibility of PUF technology in this context is performed through the design and implementation of a prototype system. By exploring the potential of PUFs in this context, the thesis aims to contribute to the development of trusted data provenance solutions extending to the IoT domain.

The work provided in the thesis includes an account of the design and implementation of the prototype, consisting of three main components. An evaluation of the security and efficiency of the prototype system is also included, exposing some vulnerabilities and potential solutions to patch these. The efficiency evaluation included concludes that the performance is adequate given the context, but also provides a possible strategy to improve sensor data throughput of the system.

In conclusion, the prototype system and work included in the thesis lays a foundation for the viability of PUF technology as a means to bootstrap the integrity of sensor data labels at the IoT device level.

Acknowledgements

I would like to thank my supervisor, Elisavet Kozyri, for the continuous guidance, advice and positive energy throughout this year. She has been the best supervisor I could ever hope for and I am truly immensely grateful for this.

I would also like to thank my co-supervisor Tor-Arne Schmidt Nordmo for sharing his valuable knowledge and insight within the IoT domain. Thank you for assisting me whenever I was struggling with hardware related issues, and always providing me with immediate alternatives.

Another thank you to Marius Ingebrigtsen for being a great friend and office-comrade throughout the year. The daily talks and moments I have had with him throughout the year have been very valuable to me, both academically and personally. I am thankful that he always (most of the time...) accepted my offerings of morning coffee from my pitiful thermos, as it gave me a moment of subtle joy every time.

Lastly, I would like to thank my mother, Hanna, for believing in me, showing unconditional love and support throughout this endeavor. She has been amazing, as always.

Contents

List of Figures

List of Tables

List of Abbreviations

AES advanced encryption standard

API application programming interface

ASCII american standard code for information interchange

CAGR compound annual growth rate

CBC cipher block chaining

CPU central processing unit

DOS denial of service

ECB electronic code book

GDPR General Data Protection Regulation

HTTP hypertext transfer protocol

I/O input/output

ID identity

IOT Internet of Things

IV initialization vector

JSON JavaScript Object Notation

LDP labeled data packet

LDPA labeled data packet array

LQI Link Quality Indicator

MCU microcontroller unit

MITM Man-in-the-middle

MQTT message queuing telemetry transport

NFC near-field communication

NTP network time protocol

PUF physically unclonable function

RAM random-access memory

RGB red green blue

SDK software development kit

SID pseudonym identity

SRAM static random-access memory

SSID service set identifier

SSK shared secret key

TLS transport layer security

TSK temporary session key

UART universal asynchronous receiver/transmitter

URL uniform resource locator

USART universal synchronous/asynchronous receiver/transmitter

USB universal serial bus

USBC universal serial bus type C

1 Introduction

Decisions made today by corporations, government entities and businesses rely increasingly on statistics and predictions derived from data. Decision-support systems are ubiquitous and are utilized at a global scale, impacting both the global economy and politics worldwide. As smart devices, sensor technology, and IOT advances, their practicality and use cases increase, and these technologies are progressively being leveraged in both legacy and new systems to provide novel services. Wearable devices equipped with sensors are already employed in many sectors such as elderly care [1] and medical monitoring [2], enabling real-time monitoring of vital signs and physical activity. Figure 1.1 illustrates the rapidly growing wearable technology market, expected to reach USD 493.26 billion by 2029, growing at a compound annual growth rate (CAGR) of 17.60% in the period 2024 – 2029.

However, in recent years, new laws and regulations such as the General Data Protection Regulation (GDPR) [4] have been introduced, imposing restrictions on both the collection and usage of personal data. Organizations and businesses are increasingly responsible for complying with these new regulations, which demands significant changes to their internal technical and logistical structures. Collecting and processing sensed user data from smart devices in a manner compliant with GDPR introduces new challenges related to security and data provenance. An automated assessment [5] of the compliance of mobile apps with the cross-border transfer requirements of the GDPR, done on the Google Play Store, notes that 48% of apps that sent personal data either completely or partially failed to comply with the regulations. Reasons for failure to comply

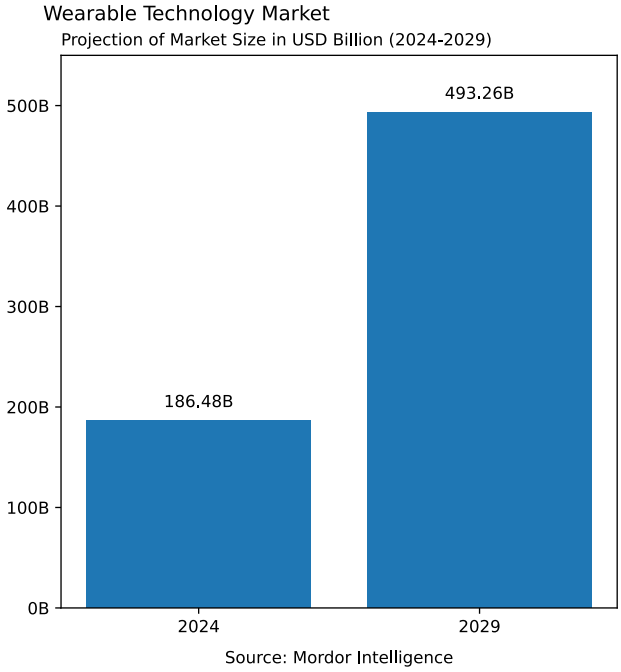

Figure 1.1: The growing wearable technology market [3]

included ambiguous or inconsistent disclosures about cross-border transfers in their privacy policies or omitting any disclosure on the topic.

Although there are numerous challenges organizations face with regards to compliance with GDPR [6], the focus of this thesis is limited to the technical side of the restrictions and stipulations where compliance is dependent on data provenance mechanisms across conventional data sources as well as extending to the diverse ecosystem of smart devices and the IOT. As an example, in the context of compliance with the right to be forgotten, a smart device user has the right to have all data associated with them to be deleted, including measurements, derived analyses and models trained with their data. In order for systems incorporating smart devices in their user data collection to be compliant with the right to be forgotten and other regulations that need data provenance, some form of end-to-end tracking of data is required, starting at the point where the data was sensed. In many cases, such capabilities in existing/legacy systems are either severely limited, unnecessarily complex or inefficient. If decision-support systems leveraging smart devices are to be viable in modern contexts where user privacy concerns are becoming paramount, streamlined data control must be incorporated into the designs of new systems as part of their core functionality.

1.1 Problem Statement

Using smart devices for user data collection processes compliant with the GDPR implies the need to collect provenance information originating at the device level. One way to address this, is to use data labeling performed at the device, where labels must include information about the ownership and origin of the data. In order to provide trusted data provenance, the integrity of these labels and their semantic connection to the data they label must persist throughout the entire life-cycle of the data, from the smart device where the sensor data was captured to the cloud and third party applications.

This integrity requirement demands security mechanisms which safeguard not only against attacks on the network level, but also against threats of physical data extraction and manipulation at the microcontroller level in smart devices.

The focus of this thesis is is related to the integrity requirement on data labels described above. The thesis is dedicated to the initial process of bootstrapping the integrity guarantees of sensor data labels generated at the device level, employing a PUF. The problem statement is formally defined below:

> *How can the integrity guarantees of sensor data labels be bootstrapped securely at the microcontroller level within smart devices? Furthermore, how can the trustworthiness of these bootstrapped integrity guarantees be ensured, especially in scenarios where devices face threats like physical extraction of sensitive data, such as encryption keys, due to deployment in vulnerable environments? Is it feasible to utilize Physically Unclonable Functions to both bootstrap and sustain the integrity guarantees of sensor data labels generated at the device, considering the aforementioned threats?*

1.2 Methodology

Using the intellectual framework for the discipline of computing [7] established by the Task Force on the Core of Computer Science, the work presented in this thesis leans towards the *design* paradigm. The framework defines the design paradigm as a four-step process followed in the construction of a system (or device) to solve a given problem, originally rooted in engineering:

1. State requirements;
2. State specifications;
3. Design and implement the system;
4. Test the system.

Starting out, the project work focused on familiarization with state-of-the-art research within acIOT, including the technologies used and micro-controllers in general. A fair amount of time was spent reading existing literature on topics relevant to the project, most notably papers related to data provenance and security challenges in IOT. The intent was to use existing research in the field to solidify the requirements for the prototype, acquire inspiration for potential implementation strategies, and to assess the potential challenges implied in the different strategies. The IOT data provenance domain and different techniques employed were mapped out according to their data labeling schemes, integrity guarantees, assumed threat models, mechanisms and hardware used.

After deciding on which data provenance technique was to be used in enabling data labeling at the microcontroller, appropriate hardware was acquired. There were several time-consuming problems and challenges in this process, as finding appropriate hardware proved difficult due to the niche use-case of PUF technology in the chosen data provenance technique.

The design and implementation of the prototype was an iterative process. Both the design and the threat model assumptions changed as the implementation progressed due to encountering hardware restrictions and limitations.

1.3 Limitations and Assumptions

1.3.1 Scope

The thesis project scope encompasses bootstrapping and maintaining the integrity guarantees of sensor data and data labels given the following context: A device captures sensor data at the microcontroller level, generates data labels for said data, encrypts a combined packet of the data along with the data labels and transmits the packet to a server.

The thesis project does not aim to guarantee the persistence of of the aforementioned integrity guarantees through any extended data pipelines which might or might not exist beyond the initial server destination. Additionally, the thesis project does not aim to guarantee or preserve any semantic connection between sensor data and its associated data labels should they at some point be separated.

While the project scope does not extend into data pipelines beyond the server (e.g., cloud processing) or offer intricate binding mechanisms between sensor data and labels, it holds research value. The work done in this thesis project can serve as a valuable asset for future projects and endeavors with a broader scope in mind.

1.3.2 Resource Constrained Components

It is assumed that the microcontrollers in the prototype system are resource constrained, impacting design choices made and protocols used.

1.3.3 Network Model

Figure 1.2 illustrates the prototype network model assumption, where dashed lines indicate wireless communication, and solid lines indicate wired communication. The prototype network consists of microcontrollers equipped with sensors and a PUF module, a wireless gateway and a server or base station.

Each PUF module and sensor has a wired connection to their own microcontroller, and the microcontrollers are wirelessly connected to the internet gateway connecting them to the internet. The microcontrollers are considered to be deployed in a physically vulnerable location and the server is assumed to be located elsewhere in a physically safe location.

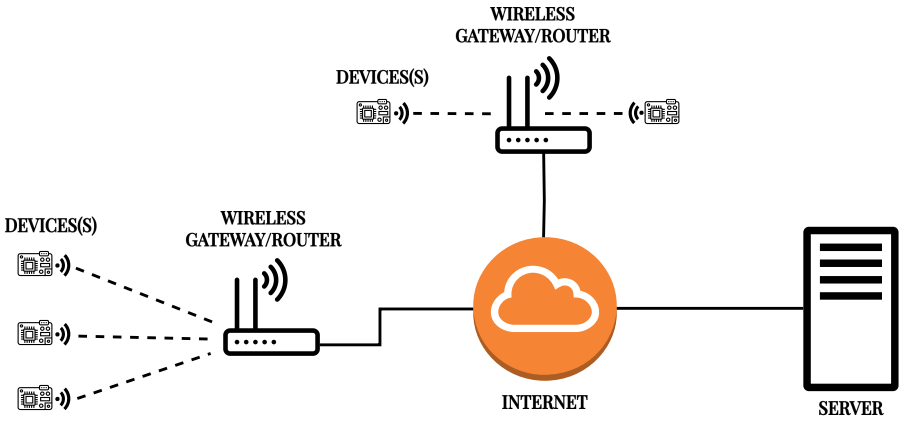

Figure 1.2: Prototype network model assumption

1.3.4 Threat Model

The threat model used for the project assumes an adversary with capabilities slightly extended beyond those of the adversary in the Dolev-Yao [8] model.

The adversary's capabilities in the Dolev-Yao model:

- **Eavesdropping:** The adversary can intercept any communication between legitimate participants in the protocol. They can listen in on the network and capture all messages exchanged

- **Message Modification:** The adversary can alter the content of intercepted messages. They can edit, delete, or even replay messages to manipulate the conversation.

- **Message Fabrication:** The adversary can create entirely new messages and inject them into the communication flow. They can forge messages pretending to be a legitimate participant.

In addition to these, the adversary is extended to also be able to:

- **Persistent Data Storage Extraction:** The adversary is able to gain physical access to IOT devices in the system and extract data from persistent (i.e., flash) storage, such as secret keys.

1.4 Contribution

The thesis project and work provided contributes a system that bootstraps the integrity of sensor data and data labels being transmitted to a safe server or trusted location. It protects secret keys and other sensitive data from physical data extraction techniques performed on persistent storage. The work provided also facilitates future work in proofing the system against physical data extraction in RAM. The work provided can also serve an implementation base for more sophisticated labeling schemes to be integrated into such systems.

The design principles and implementation of the system can be used as inspiration and/or as a component in systems outside the scope of the thesis project. An example of this is the cloud processing system published by Henze *et al.* [9]. The system proposed in this paper provides users with enforced end-to-end access control on their data using data labeling, but assumes sensor data reaches the *trust point* safely.

The system implemented in the thesis project could serve as the sub-system bootstrapping the integrity of sensor data and data labels until reaching the trust point. Figure 1.3 depicts the main idea of [9] as well as how the thesis project scope fits into the larger context.

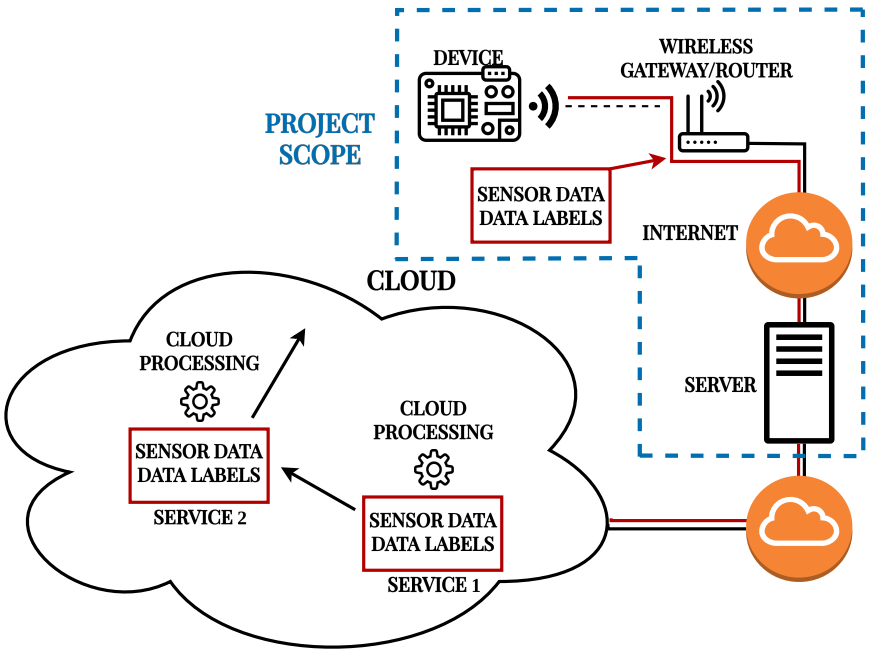

Figure 1.3: Fitting the thesis project into into a larger scope such as cloud processing.

1.5 Thesis Outline

The introduction has been focused on the what and why, the problem and the intention. The rest of the thesis is dedicated to the work which addressed the problem statement such as the design, implementation and other aspects of the prototype system.

2. **Background:** The background chapter encompasses the concepts and definitions necessary to understand the project prototype design and implementation.

3. **Design:** In the design chapter, the different components making up the prototype will be described in detail. These descriptions include the responsibilities of each component, their relationships to other components and how they communicate.

4. **Implementation:** The implementation chapter includes detailed descriptions of how the design of the prototype was implemented, including code examples and control logic.

5. **Evaluation:** The performance, security and other metrics of the prototype is evaluated in this chapter.

6. **Discussion:** The discussion chapter includes discussions on potential problems and possible improvements to the design and implementation of the prototype. The results of the evaluation will also be discussed.

7. **Future Work:** The future work section is dedicated specifically to outlining potential avenues for future research or improvements to the current prototype.

8. **Related Work:** Lastly, the related work chapter focuses on relevant and similar research from other authors in the domain.

2 Background

This chapter will provide necessary introductions to concepts referenced in the design and implementation of the project, facilitating the later chapters of the thesis.

2.1 Data Transfer

The following section explains introduces concepts and technologies leveraged in the communication between the different components in the system.

2.1.1 Serial Communication and USART

In the context of computer science, serial communication refers to the process of sending/receiving data in a sequential order, one bit at a time [10]. Figure 2.1 illustrates the difference between serial and parallel communication in terms of sending a byte. Serial communication is considered the simplest way of transmitting data from a sender to a receiver, and is preferable when transmitting long-distance due to the synchronization difficulties introduced in parallel communication.

Serial communication protocols can be divided into several categories: *synchronous*, *asynchronous* and *bit-synchronous* [10]. Synchronous transmission

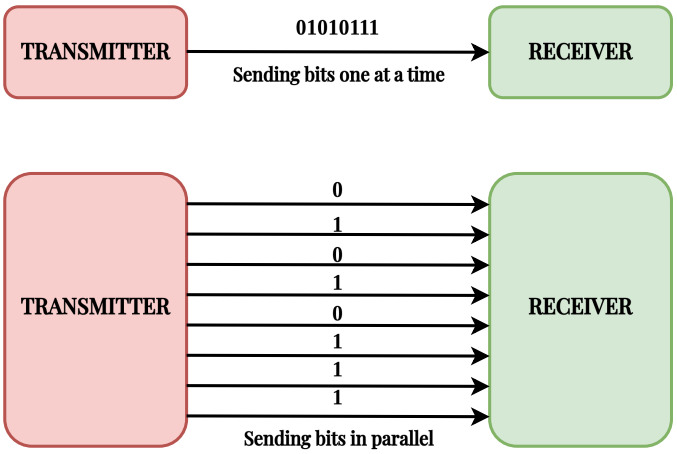

Figure 2.1: Serial and parallel transmission of a byte.

involves combining groups of bits into frames which are sent continuously, even if no data is present. In asynchronous transmission, bits are transmitted independent of the data link layer and data frames, and contain start and stop flags to allow for data gaps between frames.

USART Device

A universal synchronous/asynchronous receiver/transmitter (USART) is a serial interface device which converts received serial data into parallel data, and converts data to be transmitted from parallel data into serial data [10]. As indicated by its name, a USART device can be programmed to communicate synchronously or asynchronously (unlike UART which only supports asynchronous serial protocols). For receiving, the serial data is fetched from the serial port, converted to parallel data, and then forwarded to the central processing unit (CPU). For transmitting, the parallel data is fetched from the CPU to the USART device, converted to serial data, and then forwarded to the serial port for transmission [10].

USART Protocol

A USART device communicates using the USART protocol, which is a two-wire serial communication protocol [10]. It is a full-duplex protocol allowing for simultaneous receiving and transmitting of data, where the signal lines are labeled `Rx` (for receiving) and `Tx` (for transmitting). Note that a shared grounding cable `GND` is usually present, as USART is commonly used between devices requiring a common grounding to keep both at the same voltage. As

the clock attached to the device pulses, data is sent and received, byte by byte. As devices might not have a common clock frequency, a shared *baud rate* configured between sender and receiver determines transmission speed. A higher baud rate might result in higher data loss when transmitting through longer length cables. Figure 2.2 depicts a standard USART **asynchronous** data flow. The parity bit is used for error checking and is optional.

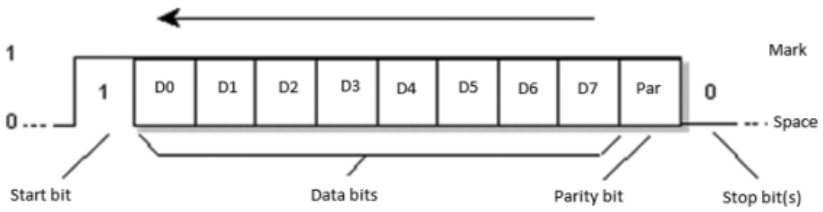

Figure 2.2: Standard USART (asynchronous)/UART data flow consisting of a byte and the start/stop flag bits [10].

2.1.2 JavaScript Object Notation (JSON)

The ECMA-404 Standard [11] defines JSON formally as:

> JSON is a text syntax that facilitates structured data interchange between all programming languages. JSON is syntax of braces, brackets, colons, and commas that is useful in many contexts, profiles, and applications.

In order to facilitate widespread interoperability, JSON is agnostic about the underlying representation of numbers and only human-readable numbers. It also offers a simple and intuitive notation for expressing collections of name/value pairs, which most languages support [11].

The most common and important features used in the JSON syntax include [11]:

- **JSON values**: A JSON value can be one of many things: an object, array, number, string, true, false, or null.
- **Objects**: An object is denoted by a pair of curly brackets encapsulating any number of name/value pairs.

- **Arrays:** An array is indicated by a pair of square brackets enclosing any number of values.
- **Numbers:** A number consists of a sequence of decimal digits without any unnecessary leading zeros.
- **Strings:** A string is a series of characters from the Unicode standard enclosed within quotation marks.

Figure 2.3 shows an example of JSON syntax in use. The `object` in this figure includes:

- Initial name/value pairs.
- A nested `object` called "batters", which consists of an `array` of `objects` called "batter".
- An `array` of `objects` called "topping".

2.1.3 Base 64 Encoding Scheme

The base 64 encoding scheme design incorporates sequences of octets which use both upper-case and lower-case letters [13]. The sequences of octets use a 65-character subset of the US american standard code for information interchange (ASCII) encoding standard, where each character is represented using a maximum of 6 bits.

The Base 64 encoding processes includes the following steps [13]:

1. Inputs are 24-bit groups, formed by concatenating 3 bytes. Output per input is a string of 4 encoded characters.
2. Encoding is done by treating the 24-bit input as 4 concatenated 6-bit groups, where each of these 6-bit groups values are mapped to a character in the 65-character US ASCII alphabet.

If the data to be encoded is not 24-bit aligned (e.g., the input is 2 bytes instead of 3), the resulting encoding will append zero padding to make up a 24-bit group. Padding in the output takes the form of a "=" character. Figure 2.4 shows the base 64 encoding scheme alphabet. Note that there also exists an alternative uniform resource locator (URL)/filename safe alphabet, which also can be found in [13].

```
{
    "id": "0001",
    "type": "donut",
    "name": "Cake",
    "ppu": 0.55,
    "batters":
        {
            "batter":
                [
                    { "id": "1001", "type": "Regular" },
                    { "id": "1002", "type": "Chocolate" },
                    { "id": "1003", "type": "Blueberry" },
                    { "id": "1004", "type": "Devil's Food" }
                ]
        },
    "topping":
        [
            { "id": "5001", "type": "None" },
            { "id": "5002", "type": "Glazed" },
            { "id": "5005", "type": "Sugar" },
            { "id": "5007", "type": "Powdered Sugar" },
            { "id": "5006", "type": "Chocolate with Sprinkles" },
            { "id": "5003", "type": "Chocolate" },
            { "id": "5004", "type": "Maple" }
        ]
}
```

Figure 2.3: Example JSON [12].

2.2 Physically Unclonable Function

As PUF technology lies at the core of the system design, a necessary introduction to the technology is provided below.

A physically unclonable function is an entity or mechanism that capitalizes on production variability in physical characteristics to produce a device-specific output, typically represented as a binary number [14]. The device-specific output of the PUF can be seen as the digital fingerprint of a device. A PUF involves the use of several components characterized by local parameter variations in a device. These discrepancies between components in a device are referred to as *local mismatches*. In order to generate the device-specific output, the components of the PUF are either directly read, combined or compared. Since the local parameter variations measured cannot be controlled in a predictable

Table 1: The Base 64 Alphabet

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/
13	N	30	e	47	v		
14	O	31	f	48	w	(pad)	=
15	P	32	g	49	x		
16	Q	33	h	50	y		

Figure 2.4: Base64 encoding scheme alphabet [13].

fashion by outside sources, a PUF is considered to be unclonable in practice. It can, however, be cloned in theory.

There are multiple approaches to creating a PUF, though most, if not all approaches depend on some signal input (i.e., a challenge) in order to generate the output. This fundamental characteristic classifies a PUF as a function. The input usually either manipulates the internal components or specifies which components to be used, resulting in an output produced by different component parameters. PUFs are usually classified by the fabric and mechanism they use, as depicted in Figure 2.5. Note that the figure does not encompass all PUF types.

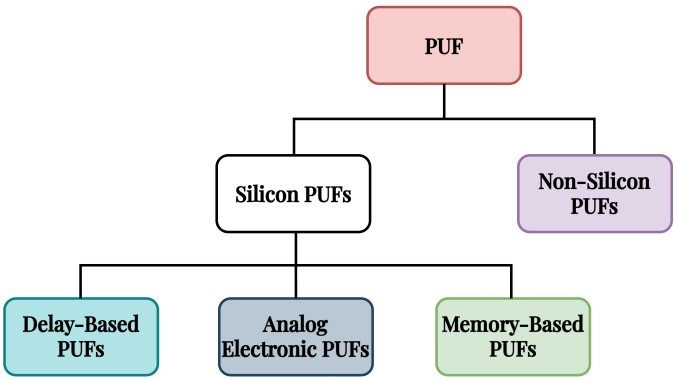

Figure 2.5: Main classifications of PUFs, where each category contains a variety of different approaches [15].

SRAM PUF

The SRAM PUF is characterized by its use of existing SRAM blocks and cells to generate memory-chip-specific data [14]. SRAMs use basic memory cells equipped with inherent feedback mechanisms, ensuring retention of stored values as long as the device is powered [16]. An example of such a mechanism is the coupling of a pair of inverters, where the output of one inverter serves as the input for the other. This is depicted in the left side of Figure 2.6, showing an SRAM bit cell retaining an asserted or non-asserted value.

Upon circuit power-up, the SRAM bit cells stabilize at a state (i.e., 0 or 1) determined by the local mismatches among the transistors involved, such as differences in transistor length, width and thickness. Consequently, each SRAM cell yields one bit of output data. Given enough SRAM cells in an SRAM array, this creates a device-specific digital fingerprint. The entirety of Figure 2.6 illustrates the process of forming a digital fingerprint using a large number of SRAM bit cells.

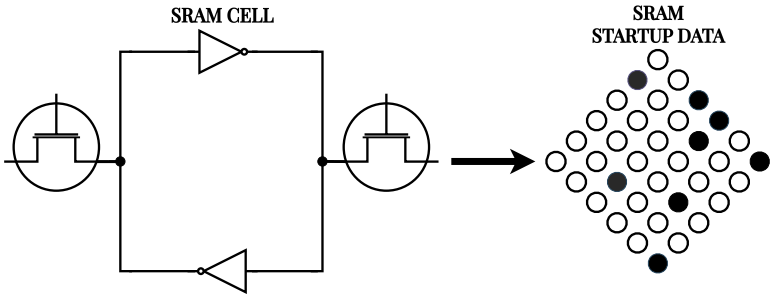

Figure 2.6: Forming a digital fingerprint using a large number of SRAM bit cells.

It is worth noting that not all SRAM implementations are suitable for PUF purposes due to insufficient mismatches between transistors.

2.3 Internet of Things

The problem statement and the work provided in this thesis lies within the IOT domain. This section defines concepts as well as technologies common within the IOT domain and provides a general understanding of the domain's architecture.

The Internet of Things is a network environment connecting a large number of heterogeneous objects [17]. Sensors, microcontrollers, microprocessors and communication technologies such as WiFi and near-field communication (NFC)

are integrated into the objects of the environment. Objects with integrated technology are referred to as IOT devices, and often include at least one microcontroller and a sensor of some kind. The IOT devices making up the environment can be connected to people, other devices and external services, providing useful data to external applications from the point of data sensing. IOT has facilitated the development of numerous applications and domains such as smart homes, smart farms and wearable health technology [17].

2.3.1 Sensors

A sensor is a device that detects stimuli or input from physical qualities and produces actionable outputs [18]. The input or physical qualities which the sensor reacts to are often referred to as *measurands*. A sensor typically comprises two components: the sensing unit, also known as the sensitive element, and a transducer, which is a device capable of converting one form of energy into another. The sensing unit interacts with the measurand and produced an output correlated with the physical quality sensed. Subsequently, the transducer converts this energy into an analog or digital signal which can be read by a data gathering system [18].

2.3.2 Microcontrollers

A microcomputer is made up of three fundamental components: A CPU, a memory unit and an input/output (I/O) system [19]. The components of the microcomputer are interconnected by electric wires called *buses*, where address buses transport memory or I/O addresses and data buses transport instructions or data. Additionally, there are control buses which transport control signals used by the different components.

Microcontrollers integrate the essential resources and components found in a microcomputer, including the CPU, memory, and I/O capabilities, into a single chip. [19]. The components of microcontrollers are resource constrained in order to minimize power consumption and space occupation, but their functionalities and responsibilities are similar to those found in a normal computing system, with some exceptions.

Considering microcontrollers are often used to provide some kind of useful data to the outside world, and this is commonly faciliated by external sensing devices, their I/O resources are crucial [19]. Some even have integrated wireless communication technology. Most microcontrollers have serial, parallel and analog ports, controlled by timers and interruption managers. In order to facilitate communication with a diverse range of external entities, microcontrollers

typically feature a high number of I/O resources relative to the available pins on the chip. Additionally, many of these pins can be configured or mapped to support various communication interfaces.

Most microcontrollers also include some form of *watchdog*, a component which monitors program execution in the microcontroller, and resets the program in case of exceptions hindering normal program execution [19]. Figure 2.7 depicts the connected components within a microcontroller.

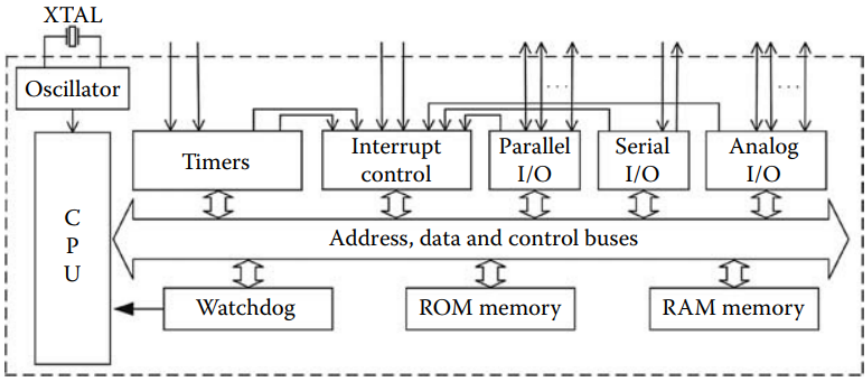

Figure 2.7: The components of a microcontroller and their interconnections [19].

2.3.3 Internet of Things Architecture

The architecture of IOT is characterized by a set of layers, where each layer is differentiated by its functionalities, incorporated devices and techniques [17]. The different layers encompassing the IOT architecture include: the perception layer, network/transport layer, processing/middleware layer and application later. The naming and granularity of these layers might differ in various literature, and some include an additional layer called the business layer. Figure 2.8 illustrates the layers of the IOT architecture.

The **perception layer** is the initial layer of the IOT architecture [17]. The layer consists of a diverse range of objects with integrated sensor and communication technology. The sensing and data gathering components of the layer are referred to as *perception nodes*. The group of components in the layer which enable data sharing, in addition to facilitating a connection to the network layer, is referred to as the *perception network*.

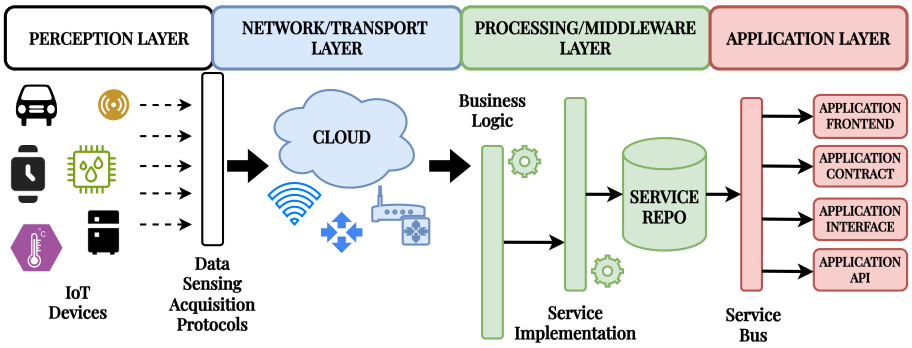

Figure 2.8: IOT architecture layers.

The **network/transport layer** facilitates the transfer of data from the perception layer to the next layer, the connection between all the IOT devices and data sharing [17]. Additionally, the components of this layer forwards data to various IOT network gateways, serving as intermediaries between multiple IOT devices to facilitate data aggregation and transfers to and from other IOT devices and networks of devices.

The **processing/middleware layer**'s responsibilities is performing the data aggregation and processing facilitated by the previous layer. In addition, the layer is responsible for composing services as well as providing these services to the next layer [17].

The **application layer**, sometimes referred to as the business layer, resides at the top of the IOT architecture and is directly accessible to end users [17]. Its main function is to oversee applications built upon data managed and services provided by the middleware/processing layer [17]. Depending on how the data is managed by the previous layer and which services are provided, this layer can be structured in many different ways.

3 Design

This chapter will first give a brief introduction to the main idea of the design and how the components and protocols are used in order to address the problem statement. Subsequently, it will go into more detailed descriptions of the components and their responsibilities, how the system is configured and provide an in-depth look at the communication protocols used. It is worth nothing that the design is inspired by Aman's paper [20] on secure data provenance in the IOT using wireless fingerprints. Figure 3.1 shows an overview of the components in the design and their connections.

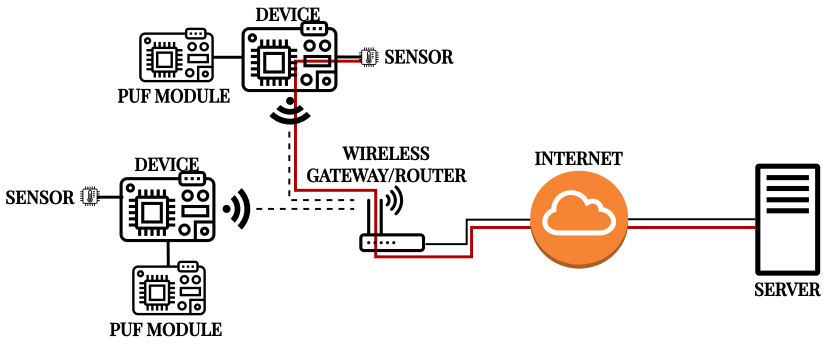

Figure 3.1: Prototype components overview. Note that this overview includes two devices. Red line indicates the data path for sensor-generated data and its label(s) from capture at the topmost device to safe arrival at the server.

3.1 Overview

The prototype system design encompasses three main components: the device, the PUF module, and the server. Although sensors and wireless gateways are technically also part of the system, they are considered arbitrary in the design and will not be discussed in detail.

The core of the system is the device, which is connected to all other components in the system. It is equipped with its own PUF module and sensor, and has wired connections to these. It is also connected wirelessly to the server through the internet. The main program flow consists of capturing and labeling sensor data on the device, encrypting it using symmetric encryption and transmitting to the server. In order to use symmetric encryption, secret keys need to be protected. The design incorporates the use of the PUF technology described in Section 2.2 in order to protect a shared secret key (SSK) assuming the threat model described in Subsection 1.3.4, where the attacker is able to extract sensitive information such as secret keys from RAM.

Safeguarding the Shared Secret Key

Considering the threat model, the primary approach to safeguarding the SSK involves avoiding its persistent storage in plain-text within the device. Instead, the SSK is encrypted by external means, and only its encrypted version is stored persistently by the device. This encryption depends on the PUF module's root key, which is a device-unique digital fingerprint generated by combining an SRAM PUF and an internal function within the PUF module.

In order to reconstruct the same root key for use in later power cycles, the PUF module uses the PUF combined with a series of bytes known as an activation code which remains stored persistently. The properties are exclusively available when the PUF module is powered on, meaning the root key cannot be extracted while the module is powered off. This approach ensures that the SSK remains secure and the root key is inaccessible unless the PUF module is activated and the system is operating.

The device component does not store the SSK in persistent memory, but relies of the PUF module to provide it when necessary. Figure 3.2 showcases the device reloading the SSK after a power cycle or similar disruption. It assumes the device is deployed and a SSK (green) has been distributed during system configuration. It also assumes the device has sent its SSK to the PUF module in order to receive an encrypted SSK.

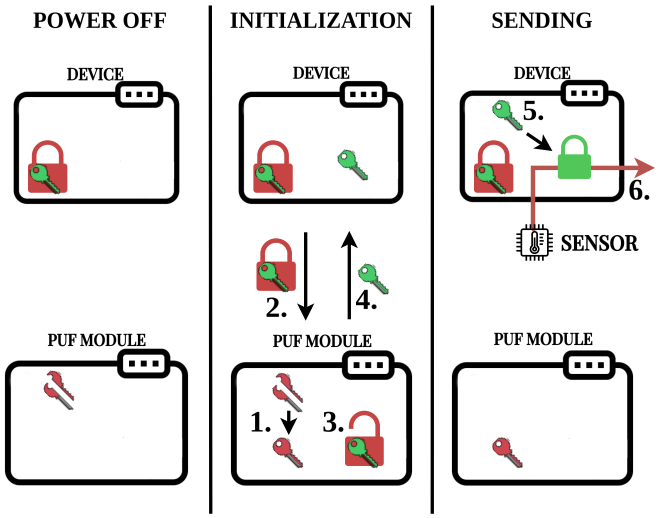

Figure 3.2: Safeguarding the SSK. Broken red key symbolizes the physical attributes and activation code. Note that the unique digital fingerprint is inaccessible during the "POWER OFF" state.

In order for the device to reload its SSK upon startup and send sensor data along with labels to the server, the following steps are executed (numbered as in the figure):

1. PUF module is powered on, and reconstructs its root key (red key) using its PUF combined with the activation code (these are symbolized as the two incomplete pieces of the red key).

2. Device sends its encrypted SSK to the PUF module.

3. PUF module decrypts the SSK.

4. PUF module sends the decrypted SSK back to the device in response to the request.

5. Device captures and labels sensor data, then encrypts the data and labels using the SSK.

6. Device transmits encrypted packet to server, which is assumed to always have access to the SSK.

The device and the server engage in a mutual authentication protocol before the transmission of sensor data starts. Data is sent using a data transfer protocol to ensure the integrity of the data packets being received at the server. These protocols will be detailed in Section 3.4.

In the context of the system design, a SSK is considered to be shared between a specific device component and the server. The server might store several SSKs, each belonging to a different device-server relationship.

3.2 Components

3.2.1 PUF Module

The PUF module serves two primary purposes: encrypting the SSK on behalf of the device and serving as a key backup mechanism should the device lose its SSK from RAM due to disruptions. As mentioned in the brief introduction, the module provides these encryption and decryption services using its own device-unique digital fingerprint, generated using PUF technology.

Digital Fingerprint Generation

The digital fingerprint (root key) is generated using the PUF module's physical properties and an internal function, and is never stored in a persistent manner in the PUF module. The generation process also yields a series of bytes known as the activation code, which, when combined with the physical properties, facilitates the reconstruction of the fingerprint. Note that the fingerprint generation should only be performed once as each generation will result in a new fingerprint and activation code. For simplicity, the fingerprint generation process will be referred to as the *Enroll* process. Figure 3.3 outlines the enrollment process of the PUF module in the design, wherein SRAM startup data and an internal function are utilized.

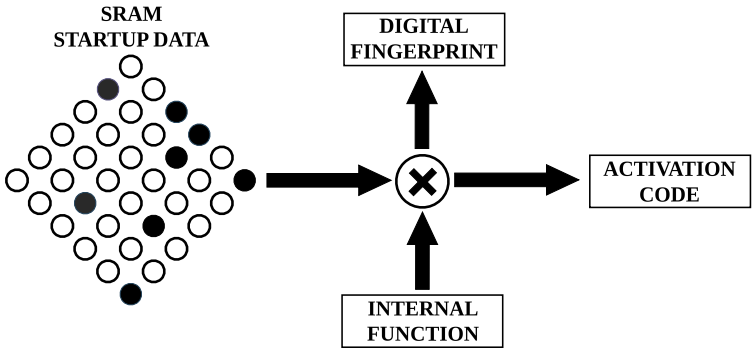

Figure 3.3: Figure showing how the PUF module generates its digital fingerprint using SRAM and an internal function, outputting the activation code [21].

Digital Fingerprint Reconstruction

In order to reconstruct the digital fingerprint, the activation code and the startup SRAM is combined as depicted in Figure 3.4.

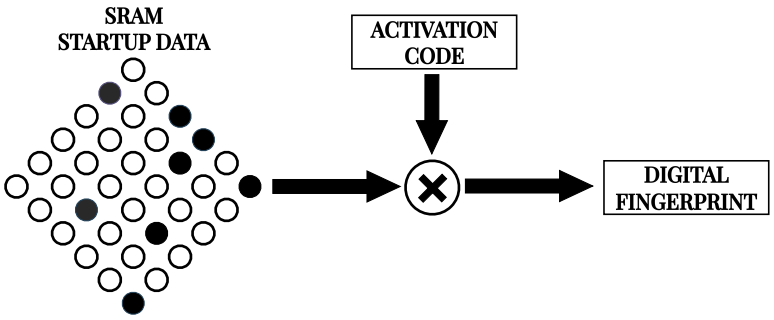

Figure 3.4: Figure showing how the PUF module reconstructs its unique digital fingerprint using SRAM and the activation code, outputting the digital fingerprint [21].

Module Operation

The PUF module offers an API to handle various requests from the device, including *Enroll*, *SetKey*, and *Challenge*. The *Enroll* service triggers the module to generate the root key and store the activation code in persistent storage for subsequent reconstruction. The *SetKey* service encrypts secret keys from incoming requests and returns the encrypted result, while the *Challenge* service decrypts keys from incoming requests and returns them in their original, unencrypted state. These services are part of a system-wide SSK configuration process, detailed in Subsection 3.3.1.

Upon startup, the PUF module will perform necessary initialization procedures and enter a perpetual server state, waiting for requests from the device component. Figure 3.5 shows the message format of communication between the PUF module and the device.

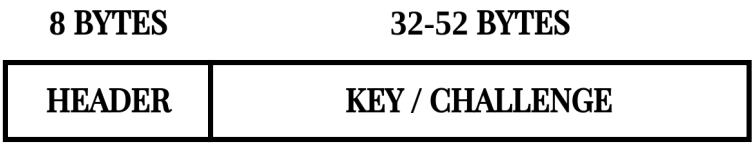

Figure 3.5: Primitive message format for communication between the PUF module and the device. The first byte of the header indicates the type of message whilst the second byte indicates the size of the message. The remaining 6 bytes of the header are reserved, but are not used in the current design.

3.2.2 Device

The device is the core of the prototype system and is either the initiator of or an active participant in all the protocols of the prototype system. The main responsibilities of the device component are receiving sensor data from the sensor, generate data labels, encrypting the data with its associated data labels and transmitting it to the server.

Devices are identified by their identity (ID) or SID depending on whether or not they have been registered, both represented as a series of bytes. SID and device registration will be detailed in Subsection 3.3.2.

API

During offline system configuration, the device is an active participant in the system-wide SSK configuration process and offers an API for managing a range of server requests. The API provides the following services: *Enroll*, *SetKey* and *VerifyKeys*. System configuration will be detailed in Section 3.3.

While the service names of *Enroll* and *SetKey* match those of the PUF module API, and both APIs serve overarching objectives, it's important to recognize that the operations performed by the device services differ from those of the PUF module. The *Enroll* service simply relays the request to the PUF module and returns the subsequent response to the server.

The *SetKey* service acquires the SSK from a server request, caches the key in RAM, and initiates device registration. Following a successful registration, it proceeds to relay the prior request to the PUF module for encryption of the key. Upon receipt of the encrypted key, the device stores it persistently and finally relays the encrypted key response to the server.

On the other hand, the *VerifyKeys* service accepts an encrypted SSK from a server request. It verifies its equivalence to the encrypted key stored in persistent flash within the device and forwards it to the PUF module for decryption. Subsequently, it verifies the received decrypted key's correspondence with the one stored in the device's RAM, and then returns the decrypted key to the server.

Device Configuration Phase

In the configuration phase the device initializes the necessary functionalities, loads data from flash (if any) storage and starts up the API server. At this

point, depending on whether or not the device is registered, it will do one of two things: if registered, the device loads its key from the PUF module and moves on to the on-line main program phase. This procedure will be described in Subsection 3.4.1. If the device is not registered, it will wait for a SSK configuration request from the server in order to begin the SSK configuration, detailed in Subsection 3.3.1. Figure 3.6 depicts the program flow of device during the configuration phase.

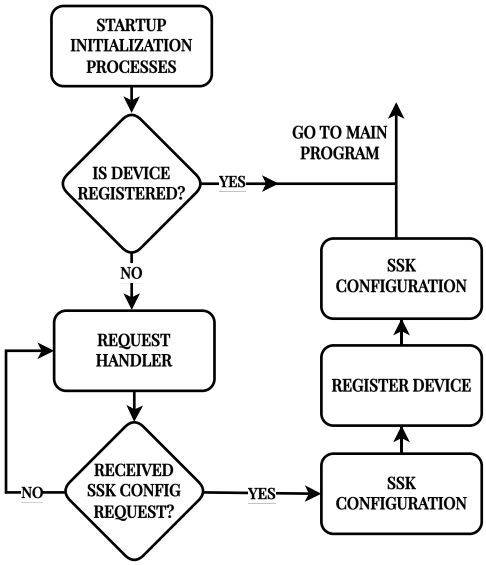

Figure 3.6: Figure depicting the program flow of configuration phase in the device component.

Main Program Loop

The main program loop is responsible for transmitting sensor data and its associated labels to the server, assuming successful completion of the SSK setup and device registration. The integrity and trustworthiness of both data and labels are ensured through two main processes.

Initially, a mutual authentication protocol is executed with the server to establish a temporary session key, distinct from the SSK. Subsequently, transmission is done using a data transfer protocol along with the established session key. Detailed descriptions of the authentication and data transfer protocols will be detailed in Subsection 3.4.2 and Subsection 3.4.3, respectively.

Note that the shared secret session key is temporary, and depending on the configuration of the system, the device will need to re-authenticate with the server frequently in order to guarantee its freshness. The logic governing the

main program loop is outlined in Figure 3.7.

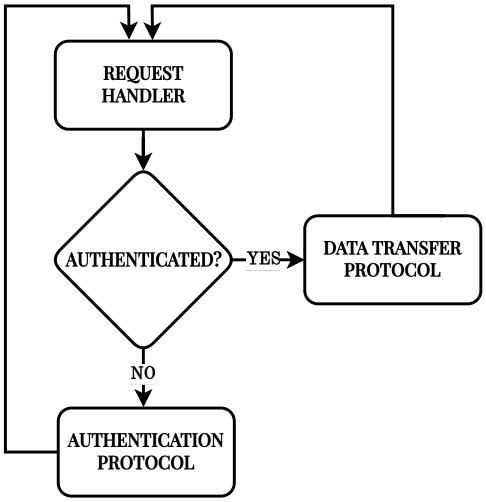

Figure 3.7: Logic governing the perpetual main program loop. Essentially a loop which handles requests and sends data or authenticates depending on the authentication status.

3.2.3 Server

The server component functions primarily as an API, with its responsibilities varying based on whether or not the system is in the configuration stage or devices have been deployed. Prior to device deployment, the server oversees external requests to commence the system-wide SSK configuration process. During this stage it also handles registration requests from devices. At this point, the server maintains a record of the devices' IDs.

After the system has been configured and devices have been deployed, the server is responsible for receiving data and labels by engaging in the authentication and data transfer protocols initiated by the devices. At this stage, the server additionally tracks each device's SSK, encrypted SSK and pseudonym identity.

3.3 System Configuration

This section will detail the system-wide SSK setup and the device registration. The wireless communication between components during these processes have the potential to reveal sensitive information such as secret keys and device identities, and must be performed offline in a safe environment. Table 3.1

contains the notations utilized in the system configuration illustrations for this section.

Notation	Description
ID_i	Identity of the device
M_i	The i-th message
N_i	The i-th nonce
SID_A^i	Pseudonym identity of device ID_A for the i-th iteration
SSK_A	Shared secret key established by the server and device ID_A
C_A	Challenge (encrypted SSK) from PUF module owned by device ID_A

Table 3.1: Notation used in the system configuration illustrations.

3.3.1 Shared Secret Key Configuration

The system-wide SSK configuration involves communication among all main components within the system, facilitated through the APIs mentioned in the preceding sections. The entire process consists of a series of API calls between the system components divided into three stages, each serving a specific purpose: The *Enroll* stage, the *SetKey* stage and the *VerifyKey* stage. Each stage is initiated by an external request received at the server. Figure 3.8 illustrates the communication between the components in the system during the different stages.

Enroll

The first stage, Enroll, serves the sole purpose of signaling an enrollment request to the PUF within the PUF module, without involving any additional procedures. The server dispatches an Enroll request to the device, which then relays the request to the PUF module. Upon receiving the Enroll signal, the PUF module proceeds to enroll its PUF and stores the resulting activation code in its flash memory for future root key reconstruction. Note that requests during this stage contain no payload.

SetKey

The purpose of the SetKey stage is to configure the SSK across the components. This entails having the secret key cached in RAM at the device, storing an encrypted version of the key in persistent flash within the device, and maintaining both versions in persistent storage at the server.

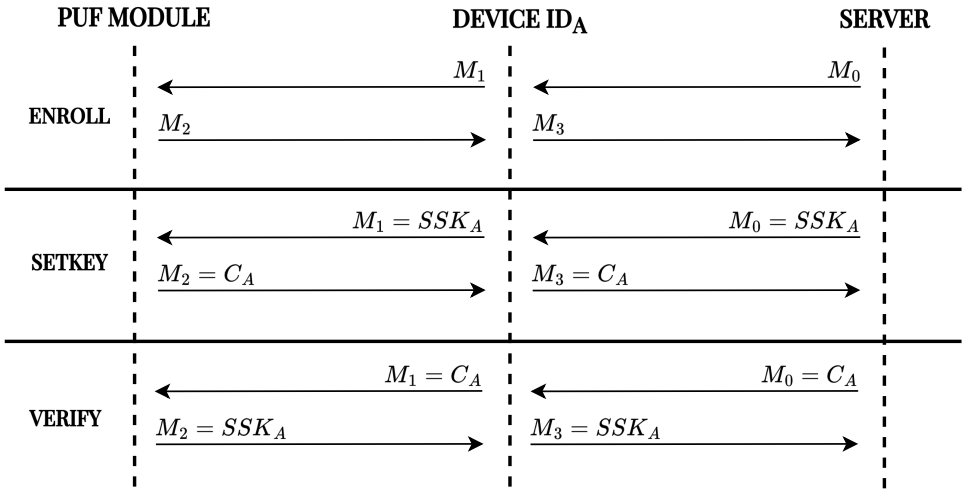

Figure 3.8: System-wide key configuration. Communication between the components during the three stages.

Initially, the server is tasked with distributing the SSK. It generates the key, preserves it in persistent storage, and dispatches a SetKey request containing the SSK to the device component.

Upon receiving the request, the device fetches the key and caches it in RAM before commencing the device registration protocol, pausing the SetKey process until registration with the server has been completed. More on this in Subsection 3.3.2. After successfully registering, the device resumes its prior operations and forwards the ongoing SetKey request to the PUF module component.

After receiving the request, the PUF module activates its PUF, reconstructs its root key and encrypts the received SSK. The encrypted key is then transmitted back to the device, where it is stored in persistent flash. Subsequently, the device relays the encrypted key to the server for permanent storage.

Note that the shared key is never stored persistently in plain-text in the PUF module nor the device, as part of compliance with the threat model in 1.3.4.

VerifyKeys

The VerifyKeys stage is executed to ensure the consistency of SSKs and their encrypted counterparts across the system components. Additionally, it validates that the PUF module returns the expected secret key when challenged with the provided encrypted key.

The verification process begins with the server initiating a VerifyKeys request to the device, containing the encrypted key. Upon receiving the request, the device performs an equality check between the received encrypted key and the one stored in persistent flash. Subsequently, the device relays a challenge with the encrypted key to the PUF module.

The PUF module decrypts the encrypted SSK using its root key and returns the resulting decrypted key back to the device. Upon receiving the decrypted result, the device performs another equality check, this time comparing its key stored in RAM with the one received from the PUF module. Finally, the device relays the response back to the server, which conducts a final equality check on the received key with the one it has stored.

If all equality checks are successful, the system confirms the consistency of both SSKs and their corresponding encrypted versions across the relevant components.

3.3.2 Device Registration

To maintain anonymity for devices across the network, the authentication and data transfer protocols leverages a constructed SID to identify the device without revealing the actual identity. Device registration in the system involves establishing an initial common SID between a device and the server. The SID of device ID_A for the i-th iteration is constructed as $SID_A^i = H(ID_A \,||\, N_a \,||\, SSK)$, where N_a is a random nonce and H is a hash function.

Given that the SID for a specific device depends on its SSK in order to be constructed, the device registration takes place during the SetKey API calls, immediately after the device receives its SSK. Figure 3.9 shows the nested communication of the registration process. Upon retrieving the SSK from the server's SetKey request, the device triggers the registration process. It generates a random nonce, constructs its SID, stores it, then forwards a registration request containing the device identity and nonce to the server.

After receiving the information from the device, the server searches for the device ID in its stored list of devices, and fetches its stored SSK for that device. The server then constructs the same SID as the device using the combined information received and its SSK, and stores it before returning the SID back to the device for verification.

Upon receiving the SID from the server response, the device verifies that both its own and the server's constructed SIDs are identical, ensuring correctness during the authentication and data transfer protocols. Once the device has

been successfully registered and the SSK setup has completed, the device is considered ready for deployment.

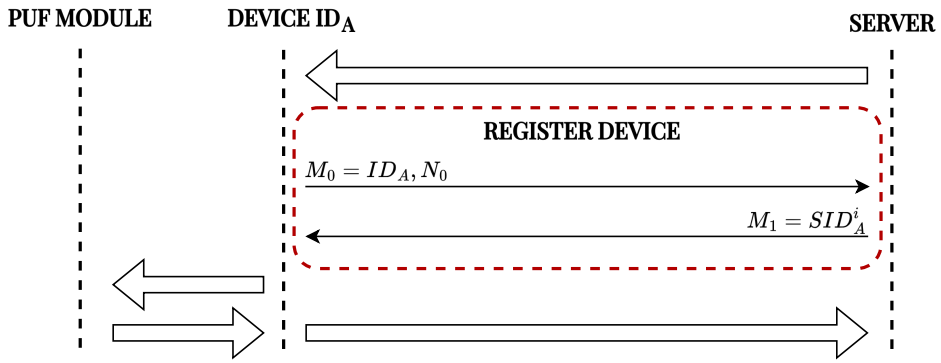

Figure 3.9: Communication during device registration nested within the SetKey API call chain.

3.4 Protocols

The following section details how the device reloads its SSK after disruption, the authentication protocol and the data transfer protocol. Expanding upon Table 3.1, Table 3.2 contains additional notations utilized in the illustrations of the protocols detailed in this section.

Notation	Description
$H(X)$	Hash of X
$\|\|$	Concatenation operator
$\{M\}_k$	Message M is encrypted using key k
I_i	The i-th authentication parameter
TSK_A^i	Temporary session key established during i-th authentication iteration
$LDPA$	Labeled data packet array

Table 3.2: Extended protocol Notations.

3.4.1 Device Secret Key Reloading

As mentioned before, to comply with the threat model outlined in Subsection 1.3.4, the device cannot store the SSK in persistent storage. However, relying solely on storing the key in RAM without a means of recovery proves inadequate. Any restart or unexpected power loss would lead to permanent loss of the secret key on the device.

In response to this challenge, the system provides a mechanism to address power-related disruptions at the device. As previously mentioned, the device stores the encrypted SSK in persistent flash storage. In the event of power loss or cycling, the device loads the encrypted SSK from flash and dispatches a challenge request to the PUF module. Upon receipt, the PUF module decrypts the challenge bytes using its reconstructed root key and returns the resulting SSK to the device. Figure 3.10 depicts the communication between the device and the PUF module during this operation.

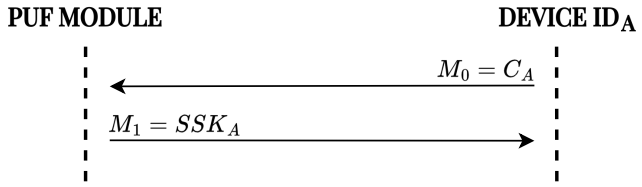

Figure 3.10: Communication during SSK reloading process.

3.4.2 Mutual Authentication Protocol

The mutual authentication protocol utilized in this design is derived from the protocol outlined in [20]. It involves two main components: the device and the server. The main end-goal of the protocol is to establish a common temporary session key (TSK) between the participating parties. The protocol is designed to be lightweight and ensure device anonymity while also enforcing authentication freshness.

As explained in Subsection 3.3.2, SIDs are utilized to maintain the anonymity of the device. Within the authentication protocol framework, each SID is valid for only one iteration. Upon successful authentication, both parties update the device's registered SID using the same computational process. The initial SID established during device registration is used in the first iteration of the authentication protocol. The protocol leverages encryption and decryption of nonces in order to prove knowledge of the SSK and utilizes *authentication parameters* in order to guarantee the integrity and correct receipt of messages between the two parties.

An authentication parameter is defined in [20] as:

> An authentication parameter is a cryptographically secure hash of a message concatenated with freshness identifiers and a secret key.

Figure 3.11 depicts the communication during the mutual authentication protocol, following the steps described below. Definitions of the messages M and

authentication parameters I are included in the figure:

1. The device generates a random nonce N_0, encrypts it using its SSK and sends the initial message M_0 along with the message authentication parameter I_0 to the server.

2. Upon receipt, the server searches for SID_A^i in its list of registered devices and fetches the appropriate SSK registered for the device. The server then uses the SSK to verify I_0 and decrypt the random nonce N_0 contained within M_0.

3. Subsequently, the server generates a second random nonce N_1, encrypts the concatenation of the two nonces $\{N_0 \mid\mid N_1\}$ using the SSK and sends M_1 along with I_1 back to the device. At this point, the server also generates the TSK where $TSK_A^i = H(N_0 \oplus N_1) \oplus H(ID_A \oplus SSK)$. Note that $\oplus$ refers to the bit-wise XOR operator.

4. When receiving M_1 and I_1 from the server, the device decrypts the encrypted concatenation of the nonces $\{N_0 \mid\mid N_1\}_{SSK}$ using its SSK. It extracts the second nonce N_1, verifies I_1 and generates TSK_i using the same computational process the server used previously.

5. At this point, both of the participants generated the TSK_A^i. They have authenticated themselves and proven their knowledge of the SSK by sending authentication parameters successfully verified by the other participant. Next, they ensure correctness of both the generated TSK_A^i to be used in this session and the SID update from $SID_A^i \rightarrow SID_A^{i+1}$ to be used in the next authentication protocol.

6. The device updates its pseudonym identity $SID_A^{i+1} = H(ID_A \mid\mid N_1 \mid\mid SSK_A)$ and sends I_2 as acknowledgement to the server.

7. Upon receiving I_2, the server generates SID_A^{i+1}, stores it for future authentication requests from the device and verifies I_2. The verification of I_2 ensured correctness of both the SID updates and the newly established TSK. Authentication is now considered complete, and the device can send data utilizing the TSK and the data transfer protocol.

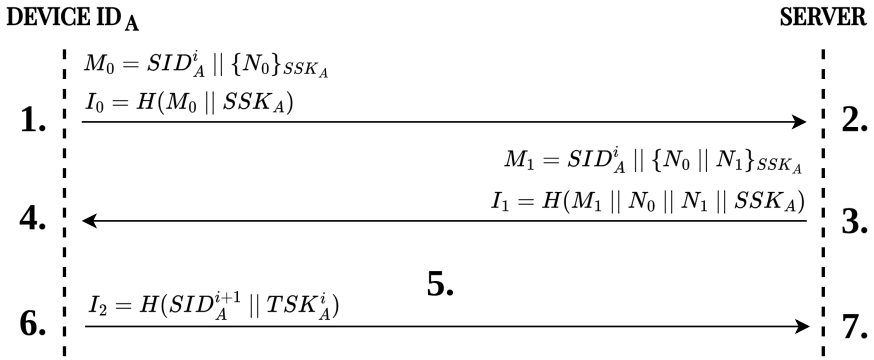

Figure 3.11: Communication between a device and the server during mutual authentication protocol.

Authentication expiration

In order to set an expiration time for the authenticated device, the server temporarily stores the SID used in the authentication (i.e., before the update) in a list of currently authenticated devices. After a certain time frame depending on the freshness requirement set at the server, the SID representing the authenticated device will be dropped from the list. The list is checked upon every data transfer request, and a device who has had its SID dropped will be forced to re-authenticate with the server.

3.4.3 Data Transfer Protocol

The data transfer protocol utilized in this design is also derived from the protocol outlined in [20], specifically the data transfer phase. The main goal of utilizing the protocol is to ensure the integrity and correctness of both sensor data and data labels being sent from the device to the server. Similarly to the authentication protocol, the data transfer protocol involves the device and server components, and also utilize a SID as the device identifier. Figure 3.12 illustrates the concept of a LDP, which is the structure used to package sensor data and its associated labels.

Note that the labeled data packets are not transmitted separately, but in batches of 10. The packets are sent in batches to reduce the communication overhead introduced by the data transfer protocol, as depicted in Figure 3.13.

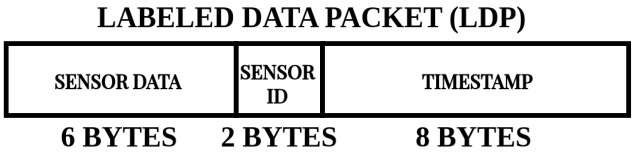

Figure 3.12: Structure of the labeled data packet.

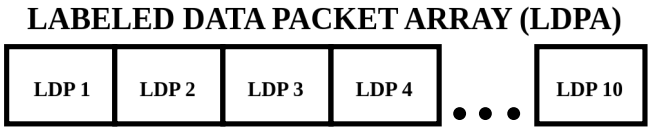

Figure 3.13: Structure of the labeled data packet array, consisting of 10 LDPs.

Figure 3.14 depicts the communication during the data transfer protocol, following the steps described below. Note that definitions of the messages M and authentication parameters I are found in the depiction as well:

1. The device captures sensor data, fetches the ID and generates a timestamp for the data. It then wraps these three components into an LDP. The device continues this process until it has 10 LDPs, constructing a LDPA for batch sending. This is done to reduce the communication overhead introduced by the data transfer protocol. Subsequently, the device generates a random nonce N_0 and encrypts the concatenation $\{LDPA \mid\mid N_0\}$ using its TSK. The device then sends M_0 and the authentication parameter I_0 to the server.

2. Upon receipt, the server uses the appropriate TSK to decrypt the payload $\{LDPA \mid\mid N_0\}$ of M_0, and verifies I_0 using the decrypted LDPA and N_0.

3. The server sends the authentication parameter I_1 back to the device as an acknowledgement of the received data.

4. The device verifies the acknowledgement authentication parameter I_1 sent by the server. If the device succeeds in verifying the acknowledgement, the current iteration of the protocol is completed and the device will continue sending new data in the same manner. If the verification fails, it will attempt to re-send the data it failed to verify from the server acknowledgement.

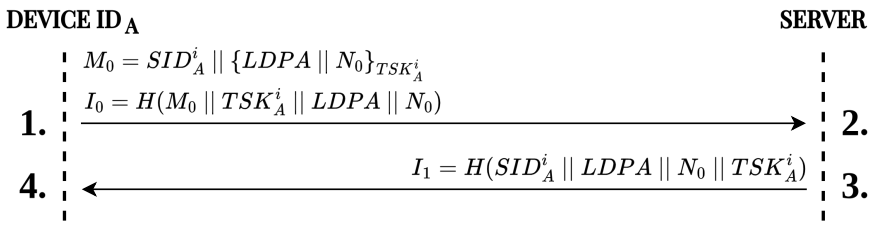

Figure 3.14: Communication between the device and the server during the data transfer protocol. Note the usage of LDPAs to send LDPs in batches.

Handling Authentication Expiration

If the device's authenticated session expires while transferring data, it will receive a 401 *Not Authorized* response from the server. It will then initiate a new mutual authentication protocol in order to create a new authenticated session.

4

Hardware and Implementation

This chapter provides details on the programming specifics and hardware used in implementing the system's design. The chapter will first provide an overview of the development boards and other miscellaneous hardware, including how they are connected to each to form the system.

Following the introductory overview, the chapter will detail the implementation specifics and hardware details concerning the PUF module, device, and server components.

4.1 Hardware Overview

Figure 4.1 shows a picture of the hardware components in the system and the physical connections between them. The server and internet gateway hardware are not depicted because they are considered arbitrary. Any computer equipped with a wireless network card and any internet gateway supporting wireless communication would be suitable.

Following the numeration in the figure, the system includes:

1. LPCXpresso55S69-EVK Development Board [22] used in the implementation of the PUF module component.

2. Arduino MKR 1010 WiFi Development Board [23] used in the implementation of the device component.

3. Adafruit APDS9960 sensor (set to red green blue (RGB) sensing mode) [24] implementing the sensor component.

4. 5-pin JST ESLOV to 4-pin JST SH STEMMA QT / Qwiic Cable [25] connecting the Adafruit APDS9960 sensor to the Arduino MKR 1010 WiFi board.

5. Male/Male Jumper Wires (75mm) [26] used for the communication between the LPCXpresso55S69-EVK and Arduino MKR WiFi 1010 boards.

6. universal serial bus (USB) 2.0 - universal serial bus type C (USBC) cable used to power and upload code to the Arduino MKR 1010 WiFi (using the server as power source).

7. USB 2.0 - USBC cable used to power and upload code to the LPCXpresso55-S69-EVK (using the server as power source).

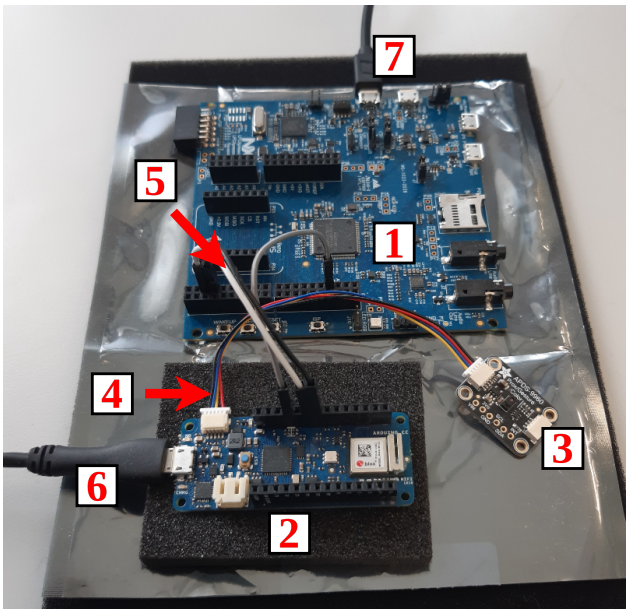

Figure 4.1: Overview of the hardware making up the system.

4.2 PUF Module Implementation

The PUF module is implemented using the LPCXpresso55S69-EVK Development Board. The 55S69-EVK was chosen due to the SRAM PUF provided with the board, as development boards providing a relatively accessible built-in PUF proved to be a scarcity. The software is written and configured using the C programming language, the MCUXpresso IDE [27] and the associated MCUXpresso software development kit (SDK) [28]. To implement the *Enroll*, *SetKey* and *Challenge* services provided by the PUF module API in the design, multiple peripherals are utilized. The API services leverage the SRAM PUF and the flash storage in their implementation, while the underlying communication providing the device component's access to the services is implemented using the USART serial protocol described in Subsection 2.1.1. The MCUXpresso SDK provides drivers and utility functions to manage the board peripherals leveraged in the programming of the implementation.

4.2.1 Board Initialization

To set the Brown-Out Detection Voltage Battery Level, which monitors the supply voltage and resets the device if it falls below a certain threshold, the following function is invoked:

```
static inline void
↪ POWER_SetBodVbatLevel(power_bod_vbat_level_t level,
↪ power_bod_hyst_t hyst, bool enBodVbatReset)
```

This is done to prevent unpredictable behaviour by the device.

Initializing Pins, Clocks and Configuring USART Pin Mapping

In order to leverage the USART peripheral for serial for communication with the device, several settings in the MCU are configured with the *MCUXpresso IDE Config Tools* [29]. Initially, the Flexcomm2 interface is enabled, which is a software-defined multi-purpose serial (e.g, SPI, I2C...) communication interface found in various NXP microcontrollers. Subsequently, the Flexcomm2 is configured to USART mode for sending and receiving data. Figure 4.2 depicts the configured MCU and Flexcomm2 pins (in green) for receiving and sending data.

In addition, the Flexcomm2 interface of the MCU is mapped to connector pins D0 (digital pin 0) and D1 on the pin expansion header of the development board, depicted in Figure 4.3. Figure 4.4 shows the usage of the mapped pins.

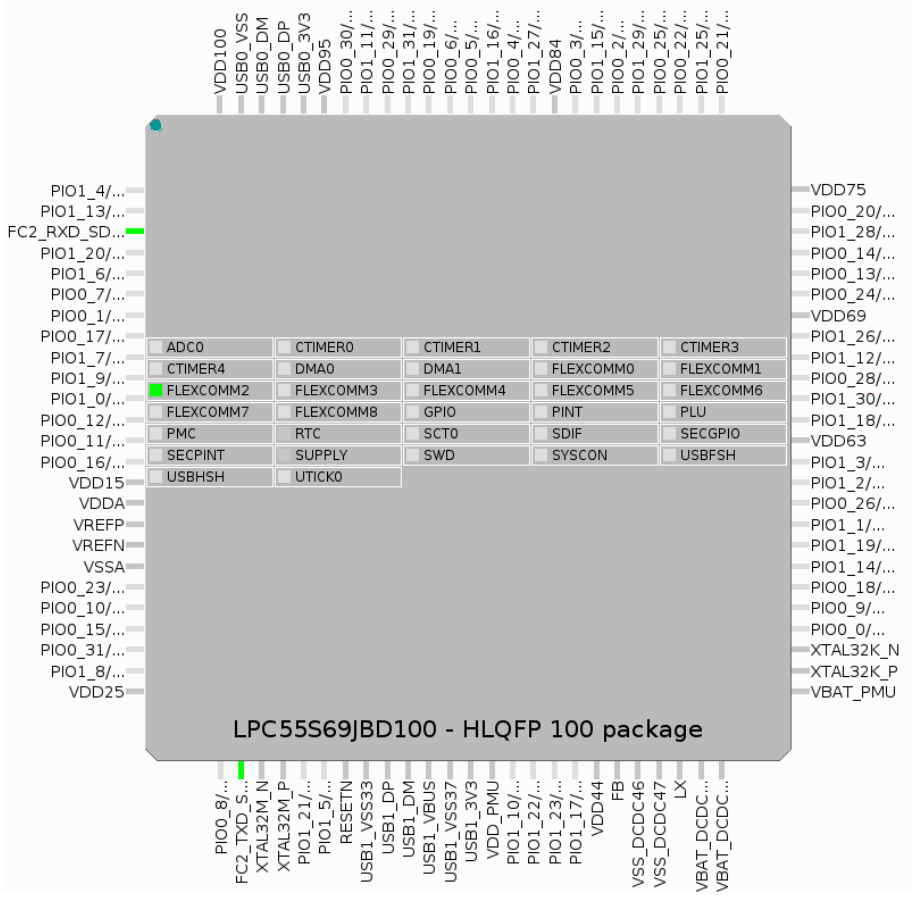

Figure 4.2: Configured MCU to enable Flexcomm2 USART using the LPCXpresso IDE Config Tools.

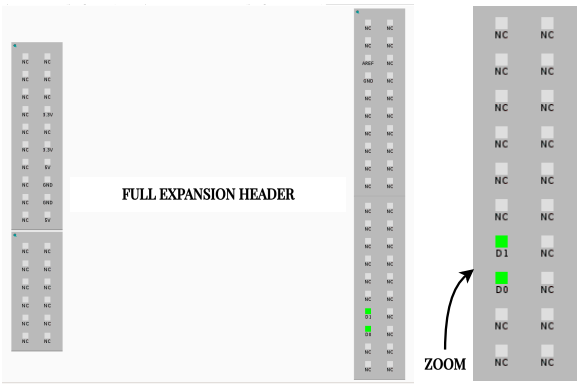

Figure 4.3: Flexcomm2 USART interface of the MCU mapped to connector pin D0 and D1 on the expansion header of the development board.

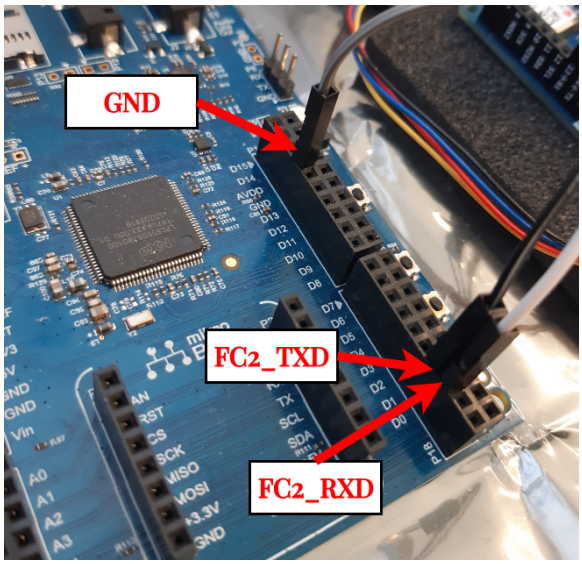

Figure 4.4: Using the Flexcomm2-mapped connector pins of the LPCXpresso55S69-EVK expansion header for USART communication.

To boot the pins and clocks necessary and apply the aforementioned mapping configurations, the following functions are invoked:

```
void CLOCK_AttachClk(clock_attach_id_t connection)
void BOARD_InitBootPins(void)
void BOARD_InitBootClocks(void)
```

`CLOCK_AttachClk` attaches the 12MHz internal clock of the board to the Flexcomm2 interface by passing the clock attach enumerator `kFRO12M_to_FLEXCOMM2` to the function. The `BOARD_InitBootPins` and `BOARD_InitClocks` functions initialize the pins and clocks of the board, including the mapping configuration applied.

Flash Initialization

To initialize the flash peripheral and load the default flash configuration the following functions are utilized:

```
status_t FLASH_Init(flash_config_t *config)
status_t FLASH_GetProperty(flash_config_t *config,
↪  flash_property_tag_t whichProperty, uint32_t *value)
```

The `FLASH_Init` function initializes the flash driver and loads the default flash configuration, whilst the `FLASH_GetProperties` is invoked multiple times to load the different properties of the flash storage into the configuration used (e.g., flash block size, flash sector size).

4.2.2 USART Initialization

The following snippet shows how the USART communication is configured:

```
void USART_GetDefaultConfig(usart_config_t *config)

config.baudRate_Bps = BOARD_DEBUG_UART_BAUDRATE;
config.enableTx     = true;
config.enableRx     = true;

status_t USART_Init(USART_Type *base, const usart_config_t
↪  *config, uint32_t srcClock_Hz)
status_t USART_TransferCreateHandle(USART_Type *base,
↪  usart_handle_t *handle, usart_transfer_callback_t
↪  callback, void *userData)
```

First `USART_GetDefaultConfig` function fetches the default configuration, followed by setting the appropriate baud rate (matching the communication target's baud rate). Subsequently, transmitting and receiving is enabled. The `USART_Init` function initializes the USART peripheral, and `USART_TransferCreateHandle` creates a handle for the initialized peripheral which is used in the USART communication programming.

Finally, the location and size of the transfer structures leveraged by the USART handle is set:

```
usart_transfer_t sendXfer;
usart_transfer_t receiveXfer;

sendXfer.data        = g_txBuffer;
sendXfer.dataSize    = sizeof(g_txBuffer);
receiveXfer.data     = g_rxBuffer;
receiveXfer.dataSize = sizeof(g_rxBuffer);
```

4.2.3 API

The main program implementing the API is built upon the `usart_interrupt_transfer` example program included in the MCUXpresso SDK, a single-threaded program which runs an infinite while loop. The loop revolves around a receive buffer `g_rxBuffer`, a send buffer `txBuffer` and four variables whose state control the USART communication: `rxOngoing`, `txOngoing`, `rxBufferEmpty` and `txBufferFull`. The following code snippet shows the four variables in use and how serial data is sent and received using the USART peripheral SDK functions:

```
while (1)
{
    if ((!rxOnGoing) && rxBufferEmpty)
    {
        rxOnGoing = true;
        USART_TransferReceiveNonBlocking(DEMO_USART,
        ↪  &g_uartHandle, &receiveXfer, NULL);
    }

    if ((!txOnGoing) && txBufferFull)
    {
        txOnGoing = true;
        USART_TransferSendNonBlocking(DEMO_USART,
        ↪  &g_uartHandle, &sendXfer);
    }

    .
    .
    .
}
```

Note that the `USART_TransferReceiveNonBlocking` and `USART_TransferSendNonBlocking` functions also invoke a callback function once their respective buffers have been either filled (send) or emptied (receive). This callback function sets the `rxOngoing/txOngoing` to false when the transmission or receiving of a message completes.

Determining Service Type

To determine which service is requested, both the message type and size of the received message is checked. Message types are indicated by the first byte in the receive buffer, while the size of the message types are static:

Enroll requests are 8 bytes (header only), *SetKey* 40 bytes (header + 32 byte key) and *Challenge* 60 bytes (header + 52 byte encrypted SSK). The `USART_TransferGetReceiveCount` peripheral function is invoked in every iteration of the loop, along with several conditional statements. The `USART_TransferGetReceiveCount` fetches the amount of bytes received from the USART handle, and saves the amount in the `receivedCount` variable. The following code snippet shows the logic of determining the type of service requested:

```
USART_TransferGetReceiveCount(DEMO_USART, &g_uartHandle,
↪  &receivedCount);
if ((receivedCount >= 8) && (header->messageType == ENROLL))
{
    DO ENROLL
}
else if (((receivedCount >= 40) && (header->messageType ==
↪  SETKEY)))
{
    DO SETKEY
}
else if (((receivedCount >= 60) && (header->messageType ==
↪  CHALLENGE)))
{
    DO CHALLENGE
}
```

Note that the `header` referred to in the code snippet is a typed `MessageHeader` pointer, pointing to the receive buffer. The `messageType` member accesses the initial byte of the message header.

Enroll

The *Enroll* service which enrolls the PUF (generates the initial device-unique fingerprint) upon a request from the device component, is implemented following the steps below:

1. The message type has been detected and necessary data has been received, so the receiving is aborted by calling the following SDK function:

```
void USART_TransferAbortReceive(USART_Type
↪  *base, usart_handle_t *handle)
```

2. The default SRAM PUF configuration is extracted into a configuration variable by invoking the SDK function:

```
void PUF_GetDefaultConfig(puf_config_t *config)
```

3. The SRAM PUF peripheral is initialized by invoking the following SDK function using the address of the PUF peripheral and the extracted default PUF configuration.

```
status_t PUF_Init(PUF_Type *base,
↪  puf_config_t *config)
```

4. The SRAM PUF is enrolled and generates the device-unique digital fingerprint by invoking the following SDK function:

```
status_t PUF_Enroll(PUF_Type *base, uint8_t
↪  *activationCode, size_t
↪  activationCodeSize)
```

   The function outputs the activation code in the `actionvationCode` pointer location, used for reconstruction of the fingerprint. This location should be a buffer with 1192 bytes reserved, to hold the activation code.

5. After enrollment, the SRAM PUF peripheral is de-initialized by invoking the following SDK function, passing its address and configuration to the function:

```
void PUF_Deinit(PUF_Type *base, puf_config_t
↪  *conf)
```

6. The destination address for the activation code in flash storage is calculated, using the flash properties in the loaded flash configuration. An example can be seen below, which calculates 3rd last page from the end of the flash storage:

```
destAdrss = flash_properties.pflashBlockBase +
↪  (flash_properties.pflashTotalSize - (3 *
↪  flash_properties.PflashPageSize));
```

7. The data at the calculated flash address location is erased using the following SDK function, to prepare for writing to the location:

```
status_t FLASH_Erase(flash_config_t *config,
↪  uint32_t start, uint32_t lengthInBytes,
↪  uint32_t key)
tatus_t FLASH_VerifyErase(flash_config_t
↪  *config, uint32_t start, uint32_t
↪  lengthInBytes)
```

8. The activation code is copied into another page-aligned (512 bytes) buffer to prepare for writing to flash, as the flash writing only supports page-aligned writes. The size of the temporary write buffer is 1536 bytes, as this is the closest page-aligned size relative to the size of the activation code. This is why the 3rd last page from the end was chosen, since 3 pages is 1536 bytes.

9. The page-aligned buffer is written to the calculated activation code destination address in flash using the following SDK functions:

```
status_t FLASH_Program(flash_config_t *config,
↪  uint32_t start, uint8_t *src, uint32_t
↪  lengthInBytes)
status_t FLASH_VerifyProgram(flash_config_t
↪  *config, uint32_t start, uint32_t
↪  lengthInBytes, const uint8_t
↪  *expectedData, uint32_t *failedAddress,
↪  uint32_t *failedData)
```

10. A response header with a length of 8 bytes is created and the initial byte is set to 1, indicating a successful enroll execution. The header is then copied into the transmission buffer `g_txBuffer`:

```
uint8_t
↪  response_header[MESSAGE_HEADER_LENGTH] =
↪  {0};
response_header[0] = 1;
memcpy(g_txBuffer, response_header,
↪  sizeof(response_header));
```

11. The program is prepared to receive another request by resetting the count of received data, zeroing the receive buffer and setting the `rxOngoing` to false.

12. The program is prepared to start transmission of the response by setting the data size of the `usart_transfer_t` send object according to the size about to be transmitted. Subsequently, `rxBufferEmpty` and `txBufferFull` are both set to true to indicate that the transmission of the response can be initiated. Transmission of the response will be initiated in the next iteration of the loop.

Setkey

The *SetKey* service which encrypts the SSK received from the device component and returns the encrypted version, is implemented following the steps below:

1. Similarly to *Enroll*, receiving is aborted by calling the following SDK function:

```
void USART_TransferAbortReceive(USART_Type
↪ *base, usart_handle_t *handle)
```

2. In order to start the SRAM PUF peripheral and reconstruct the same digital fingerprint which was generated during *Enroll*, the activation code must be loaded from flash. The activation code is read from flash by invoking the following SDK function, using the same address calculated in the *Enroll* implementation:

```
status_t FLASH_Read(flash_config_t *config,
↪ uint32_t start, uint8_t *dest, uint32_t
↪ lengthInBytes)
```

   Note that only 1192 bytes are read into the activation code buffer, even though 1536 bytes were written. Reading from flash is not restricted by page-alignment.

3. The default SRAM PUF configuration is extracted into a configuration variable and the PUF peripheral is initialized invoking the following SDK functions, similarly to steps 2 and 3 in the *Enroll* steps:

```
void PUF_GetDefaultConfig(puf_config_t
↪ *config)
status_t PUF_Init(PUF_Type *base,
↪ puf_config_t *config)
```

4. The SRAM PUF is started by calling the following SDK function, supplying the activation code read from flash. This will reconstruct the initial device-unique digital fingerprint generated in the *Enroll* service:

```
status_t PUF_Start(PUF_Type *base, const
↪  uint8_t *activationCode, size_t
↪  activationCodeSize)
```

5. The 32-byte key received through USART is encrypted using the reconstructed fingerprint by calling the following SDK function. The output location of the key is passed as the `*userKey` pointer. It's worth noting that while a key index is required to specify the PUF register for storing the key, this feature isn't utilized here. The sole objective is to return the encrypted key to the device.

```
status_t PUF_SetUserKey(PUF_TYPE *base,
↪  puf_key_index_register_t keyIndex, const
↪  uint8_t *userKey, size_t userKeySize,
↪  uint8_t *keyCode, size_t keyCodeSize)
```

6. A response is constructed. The header is created and the initial byte is set to 2, indicating a successful *SetKey* execution. Subsequently, both the header and the 52-byte key code received from `PUF_SetUserKey` is copied into the transmission buffer `txBuffer` to be sent back to the device component:

```
uint8_t
↪  response_header[MESSAGE_HEADER_LENGTH] =
↪  {0};
    response_header[0] = 2;
memcpy(g_txBuffer, response_header,
↪  sizeof(response_header));
memcpy(&g_txBuffer[MESSAGE_HEADER_LENGTH],
↪  keyCode0, sizeof(keyCode0));
```

7. The program is prepared to receive another request and start transmission of the response by following the same procedure as the last two steps detailed in the *Enroll* implementation. Transmission of the response will be initiated in the next iteration of the loop.

Challenge

The *Challenge* service which decrypts the encrypted SSK received from the device component and returns the decrypted version, is implemented following the steps below:

1. Similarly to both *Enroll* and *SetKey*, receiving is aborted by calling the following SDK function:

```
void USART_TransferAbortReceive(USART_Type
↪ *base, usart_handle_t *handle)
```

2. As in *SetKey*, the activation code is read from flash and the SRAM PUF is started by invoking the following SDK function using the same flash address:

```
status_t FLASH_Read(flash_config_t *config,
↪ uint32_t start, uint8_t *dest, uint32_t
↪ lengthInBytes)
void PUF_Setup_Start(PUF_Type *base, const
↪ uint8_t *activationCode, size_t
↪ activationCodeSize)
```

3. The 52-byte key code (encrypted key) received is decrypted using the digital fingerprint by calling the following SDK function, using the SRAM PUF peripheral address and the encrypted key code as input:

```
status_t PUF_GetKey(PUF_Type *base, const
↪ uint8_t *keyCode, size_t keyCodeSize,
↪ uint8_t *key, size_t keySize)
```

4. As in *SetKey*, a response is constructed. The header is created, this time setting the initial byte to 3, indicating a successful *Challenge* execution. Subsequently, both the header and the 32-byte key received from `PUF_GetKey` is copied into the transmission buffer `txBuffer` to be sent back to the device component:

```
uint8_t
↪ response_header[MESSAGE_HEADER_LENGTH] =
↪ {0};
response_header[0] = 3;

memcpy(g_txBuffer, response_header,
↪ sizeof(response_header));
```

```
memcpy(&g_txBuffer[MESSAGE_HEADER_LENGTH],
↪  response, sizeof(response));
```

5. The program is prepared to receive another request and start transmission of the response by following the same procedure as the last two steps detailed in the *Enroll* implementation. Transmission of the response will be initiated in the next iteration of the loop.

4.3 Device Implementation

The device component is implemented using the Arduino MKR 1010 WiFi Development board. The MKR 1010 WiFi Board was chosen due to its WiFi capabilities, easy connectivity to sensors and providing pins supporting serial communication through USART. The implementation is written in the Arduino programming language using the Arduino IDE [30]. The Arduino programming language is derived from C++ and shares its syntax.

As noted in the design section of the device component, the entire program revolves around a loop executing various tasks depending on the state of the system. The USART communication facilitating the device's access to the PUF module board's API services is implemented using the Arduino MKR 1010's native `Serial` interface. The `WiFiNINA` [31] library is employed for WiFi connectivity. Furthermore, to facilitate the device API used by the server, the `WiFiNINA` library also provides a server implementation (not the component) for handling requests on the device board. The messaging during the protocols of the design, as well as the device API services, are both implemented on top of the hypertext transfer protocol (HTTP). More on this in subsubsection 4.3.2.

Subsection 4.3.1 will detail the necessary initialization for the device implementation, while the main program and how the API is implemented will be explained in Subsection 4.3.2.

4.3.1 Configuration Phase

The device implementation requires a fair amount of initialization, both in the form of global variables and functions to be executed. This subsection describes initialization processes that are executed within the standard `setup()` function preceding the `main()` function, according to Arduino conventions.

WiFi Initialization

In order to connect to the WiFi, the network service set identifier (SSID) and password must be stored in the Arduino. This is a potential security concern which is not addressed in this project. Using the stored SSID and password, the Arduino connects to the WiFi using the `WiFi.begin(ssid, password)` function provided in the WiFiNINA library.

Initializing and Starting the HTTP Server

As noted, the HTTP server implementation is provided by the `WiFiNINA` library. The server is instantiated as a `WiFiServer` object, passing the port number as a parameter to the instantiation. Subsequently, the server is started by invoking the `begin()` method provided by the server class:

```
WiFiServer server(80);
server.begin();
```

Initializing the ADPS9960 Sensor, Real Time and the USART Interface

To initialize and configure the ADPS9960 sensor which captures the actual data, an `Adafruit_APDS9960` object is created using the `Adafruit_APDS9960` library [32]. The `begin()` function of the object is invoked, which initializes I2C and configures the sensor. The `SensorID` label used in the design is fetched from the `ID` register of the sensor.

In order to facilitate timestamp labels with correct time, the implementation leverages a network time protocol (NTP) server (pool.ntp.org) to fetch the current time. This is done since the Arduino has no concept of real time, and only calculates the time since the board started running the program.

The USART serial interface and port is initialized by invoking `Serial1.begin()`. `Serial1` refers to pins 13 (Rx) and 14 (Tx) on the Arduino MKR 1010 WiFi.

Identity and State Initialization

In order for the device implementation to communicate correctly with the server and participate in the various protocols detailed in the design, it must initialize information related to its identity and its state upon startup.

Upon startup, the device initializes its device ID, which is a 32-byte `uint_8` array, where each byte is predetermined. The device also initializes both the `registered` and `authenticated` booleans to false, indicating the default state of the device. The program will try to restore state of the SID and encrypted SSK upon startup, a process detailed subsubsection 4.3.1. In case of a fresh program upload to the board, these values are defaulted to empty values.

Recovering From Disruptions Using Flash Storage

Upon disruptions such as power loss or a power cycle, the device implementation loads the necessary recovery data from flash storage. The flash storage of the Arduino is not intended to be used for storage of information by the programming running on the board. Since the device implementation requires persistent flash storage in order to load identity-related information upon startup, a work-around is used. The implementation utilizes the `FlashStorage` Arduino library, which allows the programmer to reserve flash sections for storage of specific structures by the program.

Entering the `setup` function upon startup, the device will load data from the reserved locations in flash, checking if there is a saved SID. If there is a SID stored in flash, the device assumes its been previously registered and proceeds to set its `registered` state to true. Subsequently, it checks for an encrypted SSK in its flash storage. Should the program find an encrypted SSK, it invokes the loading of the decrypted key by sending a USART request to the PUF module board, along with the encrypted key. Once the key is received from the PUF module board, the device enters its main loop and attempts to authenticate using the SID loaded from flash storage.

Specific structures are also defined in order to hold the SID and encrypted SSK, so they can be stored in flash along with their states using the `FlashStorage` library:

```
typedef struct {
  uint8_t challenge[CHALLENGE_SIZE]; // Encrypted SSK
  bool initialized;
} challenge_t;

typedef struct {
  uint8_t pseudonymId[ID_SIZE]; // SID
  bool initialized;
} pseudonymId_t;
```

The flash regions are reserved using the `FlashStorage()` function in the following way, provided by the `FlashStorage` library:

```
FlashStorage(flashChallengeReserved, challenge_t); //
↪ Reserved region for encrypted SSK.
FlashStorage(flashPseudonymIdReserved, pseudonymId_t); //
↪ Reserved region for current SID.
```

Awaiting Configuration Request

Following the necessary initialization procedures, the program enters a loop checking whether or not the device is registered, in practice stalling until its been registered. If the device is registered, the program bypasses this loop and starts executing the main program immediately. If the device is not registered, the loop will continue to iterate until its been registered by means of a *SetKey* call chain which invokes the device registration. The following code snippet shows how the program handles the registration stalling in `setup`, as well as the main `loop` detailed in Subsection 4.3.2.

```
void setup() {
  .
  ... // Initialization procedures
  .
  while(!registered) {
    HTTPRequestHandler(); // Wait for configuration request.
  }
}

void loop() {
  HTTPRequestHandler();

  if (authenticated) {
    sendSensorData(); // Data Transfer Protocol.
  }
  else {
    performAuthenticationProtocol(); // Mutual authentication
  }
}
```

4.3.2 Main Program and API

The main program loop consists of the functions `HTTPRequestHandler`, `sendSensorData` and `performAuthenticationProtocol`. Note that this loop is single-threaded since the Arduino does not support multi-threading. Every iteration, the `HTTPRequestHandler` polls for received requests from the server component, facilitating the underlying communication for the API services. The `performAuthentication` function invokes the mutual authentication protocol, while the `sendSensorData` function invokes the data transfer protocol.

Protocol Messages Using HTTP

In order to facilitate the protocols in the design over HTTP, the `base64` and `ArduinoJson` libraries are leveraged to first encode the binary protocol messages to base 64 strings and then embed the encoded strings as JSON strings in the bodies of the requests. This is done both to ensure the correctness of the binary data being sent and to simplify the separation of the messages and authentication parameters. Using the data transfer protocol as an example, Figure 4.5 shows how M_0 and I_0 go through this process. Receiving messages go through the reverse process: unpacking the body and decoding from base 64 strings back to binary arrays. The messages are encrypted using the AES256 implementation from the `Arduino Cryptography Library` [33].

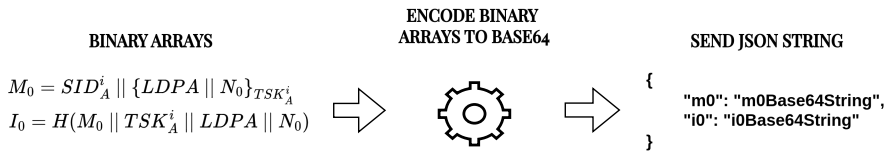

Figure 4.5: Binary arrays encoded to base 64 strings before being sent.

Accessing the API Services

The `HTTPRequestHandler` function implements the communication required to access the API services and invokes the specific service requested. Initially, a check is made to determine whether or not a new client (the server component) has attempted to connect. If a connection is detected the request received is read, and based on the request data, the appropriate API service is invoked. The following code shows the implementation details of the `HTTPRequestHandler` function. Note that once the *SetKey* service has completed, the device changes its `registered` state from `false` to `true`, as the registration process will have completed within that service:

```
void HTTPRequestHandler() {
    // listen for incoming clients
    WiFiClient client = server.available();
    if (client) {

        if (client.connected()) {
          String request = client.readStringUntil('\r');

          if (request.indexOf("GET /enroll") != -1) {
            handleEnrollRequest(&client);
          }
          else if (request.indexOf("POST /setkey") != -1) {
            handleSetKeyRequest(&client);
            registered = true;
          }
          else if (request.indexOf("POST /verifykeys") != -1)
          ↪  {
            handleChallengeRequest(&client);
          }
          client.stop(); // Close connection
        }
    }
}
```

Enroll

The `handleEnroll` function implements the *Enroll* service. Initially upon a request from the server, it creates an 8-byte header and sets the first byte to `1` (Enroll). This 8-byte header constitutes the entire request in the *Enroll* service. Subsequently, it leverages the `Serial1` interface of the Arduino to send the request to the PUF module board.

Following the request sent, the service will wait for a response from the PUF module board. It does this by continuously polling for serial data from the USART data using the `USARTReceiveHandler` in a while loop, checking for the `USART_ENROLL_SUCCESS` return value from the function. The code snippet below shows how the `USARTReceiveHandler` function is utilized in the *Enroll* implementation:

```
/* Receive USART response with a timeout mechanism. */
unsigned long timeout = millis() + 10000;  // 10-second
↪  timeout
while (USARTReceiveHandler(usartReceiveBuffer, receivedChunk,
↪  &usartTotalReceived) != USART_ENROLL_SUCCESS) {
    if (millis() > timeout) {
      // Handle timeout error
      Serial.println("Timeout error while waiting for
      ↪  USART_ENROLL_SUCCESS");
      exit(1);
    }
}
```

Initially the function checks whether or not serial data has been received by invoking `Serial1.available()`. If bytes are available, the function reads the data into the `receivedChunk` temporary buffer, before copying the read data into the `usartReceiveBuffer` which stores the entire message. Subsequently, it will increment `usartTotalReceived` according to the data read. The function will continue to iterate, read data and copy to the message buffer until the following condition has been met: a total of 8 bytes have been read, and the initial byte of the message (header) is 1, signaling an *Enroll* response. Finally, the a success-response is send back to the server component who initiated the request.

Note that the `USARTReceiveHandler` function and its usage is common for between the service functions, with slight differences. The code below shows the conditional statements used within the function, determining the return value. Note that a return value of `USART_READING` will cause the iteration of the outer while-loop to continue, invoking the function again.

```
    // Received ENROLL response
    if (*totalReceived >= 8 && usartReceiveBuffer[0] ==
    ↪  ENROLL) {
      *totalReceived = 0;
      return USART_ENROLL_SUCCESS;
    }
    // Received SETKEY response
    else if (*totalReceived >= 60 && usartReceiveBuffer[0] ==
    ↪  SETKEY) {
      *totalReceived = 0;
      return USART_SETKEY_SUCCESS;
    }
    // Received CHALLENGE response
```

```
else if (*totalReceived >= 40 && usartReceiveBuffer[0] ==
↪  CHALLENGE) {
  *totalReceived = 0;
  return USART_CHALLENGE_SUCCESS;
}
else {
  return USART_READING;
}
```

SetKey

The `handleSetKey` function implements the *SetKey* service. Starting out, the device extracts the SSK from the server request using the base 64 strategy explained in subsubsection 4.3.2.

Following the storage of the SSK in RAM, the device registration process is initiated by invoking the `registerDevice` function. To send the device registration request from the device and handle the response from the server, the `ArduinoHttpClient` [34] library is leveraged. This library builds upon the `WiFiNINA` library, and creates an `HttpClient` type by wrapping the `WiFiClient` type from `WiFiNINA` in the following way:

```
WiFiClient wifi; // WiFiNINA client type
HttpClient mkrHttpClient = HttpClient(wifi, serverAddress,
↪  port); // Wrapping the WiFiNINA type.
```

Following the completion of the device registration, the device forwards the SSK to the PUF module board using the same USART communication strategy explained in the *Enroll* implementation.

Upon receiving the 52-byte encrypted SSK from the PUF module board through the USART, it is stored in RAM and then in flash storage using the `FlashStorage` library:

```
flashChallengeReserved.write(challenge); // Writing
↪  encrypted SKK to flash.
```

Finally, the encrypted SSK is embedded in the body of the *SetKey* request response which is returned to the server using the same base 64 strategy from subsubsection 4.3.2.

VerifyKeys

The `handleSetKey` function implements the *VerifyKeys* service. Device extracts the encrypted SSK from the server request, again using the strategy detailed in subsubsection 4.3.2.

If the encrypted SSK is not present in the device, it will load it using the key loading strategy detailed in subsubsection 4.3.1. Subsequently, the received encrypted SSK from the server is compared with the one present in the device using the standard `memcmp` function. Following this procedure, it forwards the encrypted SSK to the PUF module board by means of a USART request leveraging the strategy as the *Enroll* implementation.

Upon receiving the 32-byte decrypted SSK from the PUF module board, it is compared with the key in RAM using the same `memcmp` function. Finally, it returns the `success` response to the server.

4.4 Server Implementation

The server component is implemented as a .NET Controller API running on a desktop machine, using the C# programming language. The language and framework of the server component is arbitrary as long as it supports HTTP. To implement the hashing and cryptography needed for the protocols in the design, the `System.Security.Cryptography.SHA3_256` and `System.Security.Cryptography.Aes` classes from the .NET Core ecosystem are employed. The advanced encryption standard (AES) encryption is set to electronic code book (ECB) mode using a key size of 256 bits.

4.4.1 API

To implement the necessary design elements of the server component, the .NET API employs three controllers: the `CommandController`, the `Authentication-Controller` and the `DataController`.

The `CommandController` contains endpoints related to system configuration and accepts requests from the programmer to start configuration. The `AuthenticationController` is composed of reactive endpoints which respond to device requests part of the mutual authentication protocol. Finally, the `DataController` is also reactive and responds to device requests part of the data transfer protocol.

The endpoints encompassed within the `AuthenticationController` and `DataController` exchange messages with the device in the same manner described in subsubsection 4.3.2.

4.4.2 Storing Device Information

The server implementation stores the information about devices in a JSON file. The file is made up of an array of JSON objects, each representing a specific device. The JSON approach is used due to the low demand of persistent data storage at the server.

The following snippet shows the representation of a specific device in the JSON file. Note that SSKs and SIDs are encoded from binary to base 64 before being stored in the file, since JSON only supports text format:

```
[
  {
    "id": "s3Uzx3LiE3p3WJcKcJ7MFjFDyH3LqmdUlfS/QJHog\u002Bs=",
    "pufChallengeBase64":
    ↪  "AAEABBcUfW6QvmtOpGmhCyVXJirUtrX5sfG//g/mh4fxMMK0wOYmk
    8YVrN0yWGN0aGMsNA==",
    "pufKeyBase64":
    ↪  "AQIDBAUGBwgJCgsMDQ4PEBESExQVFhcYGRobHB0eHyA=",
    "pseudonymId":
    ↪  "rsvxPME4sWxtir\u002BDlV3\u002BPX19hFn2OgSsjs/B2KSMnaQ=",
  }
]
```

4.4.3 Managing Authenticated Devices

The `Microsoft.Extensions.Caching.Memory.IMemoryCache` interface from .NET Core is leveraged to keep track of authenticated devices at the server. This interface provides the programmer with a memory cache in which data can stored. Following a successful authentication, a device's SID along with the constructed TSK for the session are stored temporarily as key-value pairs. Using this cache, the server can consult the cache with the received SID to check if the device is authenticated.

In order to store the key-value pairs in the memory cache, the `MemoryCache.Set` method is used along with a sliding timer argument. The timer argument specifies when a key-value pair is to be removed from the cache, invalidating the authentication and forcing the device in question to re-authenticate.

5 Evaluation

This chapter provides an evaluation of the design and implementation of the system. Initially, the performance and efficiency of the system will be evaluated. Subsequently, an evaluation of the system's security will be presented.

5.1 Experimental Setup

The experimental setup consisted of the components with the exact connections described in Section 4.1 and a server implementation being run on a machine with the following relevant specs:

- CPU: Intel® Core™ i7-10700 CPU @ 2.90GHz × 16
- Network controller: Intel Corporation Comet Lake PCH CNVi WiFi
- Network controller sub-system: Intel Corporation Wi-Fi 6 AX201 160MHz

During the evaluation, the system was operating at the University of Tromsø, approximately 4 meters from a wireless router connecting the `Arduino MKR 1010 WiFi` and the server machine.

5.2 Performance and Efficiency Evaluation

5.2.1 Methodology

Considering that the device component is resource-constrained, single-threaded and performs the procedures sending LDPAs, the system's performance largely depends on the `Arduino MKR 1010 WiFi` program performance. The performance and efficiency evaluation is therefore focused on the `Arduino MKR 1010 WiFi` program.

In order to evaluate the efficiency, various sections of the program were measured using the `millis` Arduino function, as well as the HTTP communication overhead in the relevant sections. The sections of the program deemed most relevant for efficiency testing were the `performAuthenticationProtocol` and `sendSensorData` functions, as these are continuously being invoked to re-authenticate with and send LDPAs to the server. The program time elapsed of the sections were measured 1000 times and then averaged to account for variability.

In order to measure the throughput, the average amount of `sendSensorData` requests the program completes over a 1 minute interval were measured. These measurements included an initial iteration of the authentication protocol, as the authentication was set not to expire during this test.

As an interesting addition, the `handleEnrollRequest`, `handleSetKeyRequest` and `handleVerifyKeysRequest` functions were also measured. These functions were only measured 10 times, providing the average elapsed time in the results table. Note that the `handleSetKeyRequest` function also involves the nested `registerDevice` function, where the measured HTTP overhead included in the results table is from the protocol in `registerDevice`.

Finally, the recovery mechanism detailed in subsubsection 4.3.1 was tested. This test was performed by removing the power source of the `Arduino MKR 1010 WiFi`, and then plugging it back in after a short delay.

5.2.2 Results

The following table shows the resulting averages of the relevant operations in the system:

Operation	Average Time (ms)	HTTP Overhead (ms)
performAuthenticationProtocol	18	291
sendSensorData	120	130
handleEnrollRequest	87	N/A
handleSetKeyRequest	2113	295 (registerDevice)
handleVerifyKeysRequest	1091	N/A
Throughput (Labeled Data Packets per min)		
2300		
Recovery Mechanism Test		
Success ✓		

Table 5.1: Measurement results using `millis()` on the `Arduino MKR 1010 WiFi`.

A brief discussion on these measurement results and a strategy for improving throughput of LDPAs will be provided in Section 6.2.

5.3 Security Evaluation

This section presents a security evaluation of the system. Note that the evaluation is performed under the assumption that an adversary will have the capabilities defined in Subsection 1.3.4. Section 6.1 includes further discussions on addressing vulnerabilities exposed in the security evaluation.

5.3.1 Methodology

The system's security was assessed by systematically examining its defenses against each individual capability possessed the adversary, performing the following steps for each capability:

1. Specifying the adversary capability against which the system is evaluated along with the system's defenses.

2. Simulating the attack or hypothesize an attack-scenario.

3. Evaluating the system's defense effectiveness in response to the attack.

5.3.2 Evaluation

Eavesdropping Security Evaluation

The following adversary capabilities are noted in the eavesdropping part of the threat model:

- The adversary can intercept any communication between legitimate participants in the protocol.
- They can listen in on the network and capture all messages exchanged.

Measures/Defense Mechanisms

- All messages and authentication parameters are encrypted using the AES algorithm with a key size of 256 bits.

Simulation:

To simulate this attack, the *Wireshark* [35] network packet analyzer was leveraged to sniff packets being sent between the device and the server in the local `uit-conference` wireless network. The intention of the packet-sniffing was to determine if an attacker could capture and make sense of the data being sent.

Result:

Using Wireshark to sniff a sensor data transfer message from the device to the server yielded the information in Figure 5.1. Note that since the data is encoded in base 64, a proper visualization of the decipherability of the raw data is not present. As seen in the figure, an adversary is able to see JSON names, but the values included are indecipherable.

Vulnerabilities Noted:

Using JSON to wrap the messages and authentication parameters exposes a vulnerability. It gives an attacker information about the intention of messages, even if the messages themselves are indecipherable. The fact that the ECB mode of AES is utilized for the encryption also creates another attack vector, since any equal plain-text will be encrypted to an equal cipher-text. These issues will be addressed in Section 6.1.

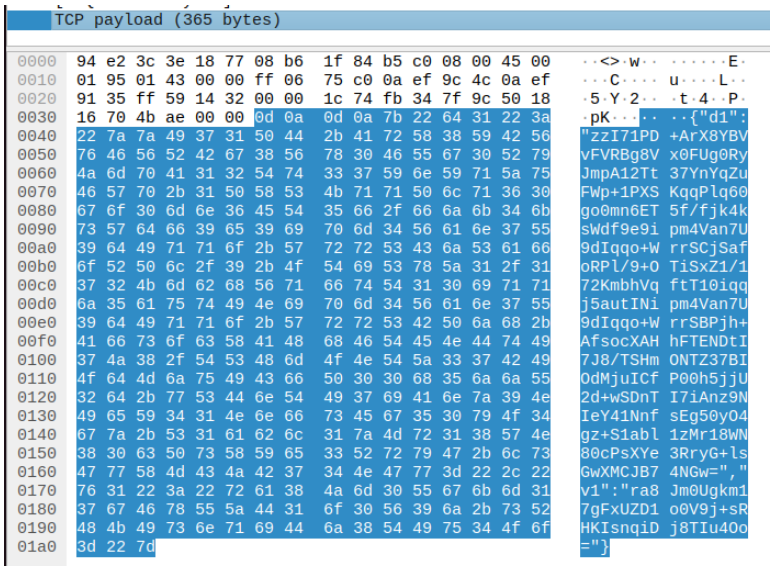

Figure 5.1: TCP payload (in blue) from wireshark packet sniffing.

Message Modification Security Evaluation

The following adversary capabilities are noted in the **message modification** part of the threat model:

- The adversary can alter the content of intercepted messages. They can edit, delete, or even replay messages to manipulate the conversation.

Measures/Defense Mechanisms:

- In order to protect against message modification attacks, all data sent between the device and server are verified using the message's associated authentication parameter.

- Messages not received by the server will cause the device to re-send the message until acknowledgement is received from the server, and the device is able to verify the acknowledgement using the associated authentication parameter.

Simulation:

To simulate this attack, messages were intentionally edited and deleted at the device before being sent to the server. Messages were also replayed without being altered.

Result:

Editing the first byte of the message payload being sent from the device resulted in the following verification failure when receiving at the server:

```
Received ReceiveSensorDatarequest from Arduino.
fail: server.Services.IdentityService[0]
SensorDataService.HandleReceiveSensorData():
↪  Authentication parameter generated for verification
↪  did not match v1 received from Arduino, cannot verify
↪  that message integrity is intact.
```

For an attacker to successfully perform this attack given the defenses in place, the attacker would first need to manipulate the message and then generate a new authentication parameter which matches the new content of the altered message. Doing this in a reasonable amount of time would require knowledge of the TSK (or SSK depending on communication type), which is never transmitted through the communication channel after device deployment.

The system is considered to be safe against most message modification, with the **exception of the scenario described below**.

Vulnerabilities Noted:

The acknowledgement from the device to the server in the mutual authentication protocol signals the server to authenticate the device in question. The acknowledgement also prompts the server to update its registered SID for the device, as a necessary step for the next authentication. Should this message be modified and cause verification at the server to fail, the device will update its SID, while the server will never receive its cue to authenticate the device and update the device SID in its list of devices.

This scenario cause two major problems: the authentication of the device will fail, **but most importantly, the device will no longer be able to authenticate, given that the SIDs of the device and server are no longer synced**. Note that this vulnerability also applies to denial of service (DOS) attacks. Figure 5.2 illustrates the scenario.

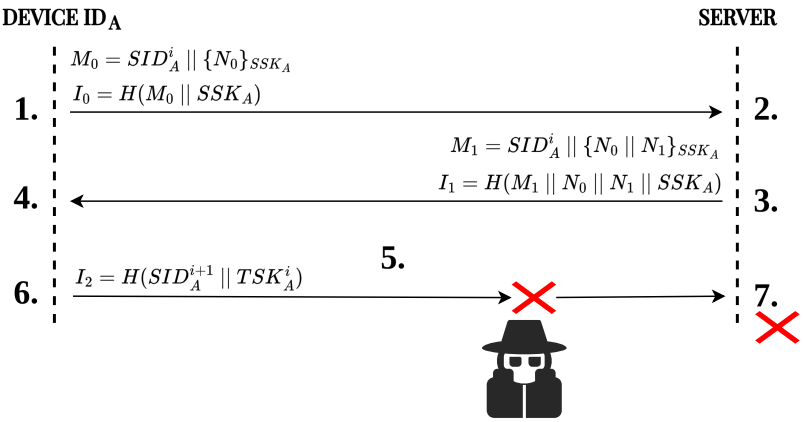

Figure 5.2: Adversary modifying/DOS-ing last acknowledgement from device to server. Note that step 7 at the server will not proceed as normal, and the server will not update its registered SID for the device.

Message Fabrication Security Evaluation

The following adversary capabilities are noted in the **message fabrication** part of the threat model:

- The adversary can create entirely new messages and inject them into the communication flow. They can forge messages pretending to be a legitimate participant.

Measures/Defense Mechanisms:

- In order to protect against message fabrication attacks, all data sent between the device and server are verified using the message's associated authentication parameter.

- Messages not received by the server will cause the device to re-send the message until acknowledgement is received from the server, and the device is able to verify the acknowledgement using the associated authentication parameter.

Simulation:

This attack was not simulated. However, a theoretical scenario can be discussed.

Result:

In order for an adversary to fabricate a legitimate message that also passes the verification check using the authentication parameter, the legitimate message would have to be fabricated using the SSK/TSK. These keys are never transmitted through the communication channel after device deployment. Considering this, any messages posing as legitimate by following the message format is detected at the receiving end by verifying the message contents using the authentication parameter.

Persistent Data Storage Extraction Security Evaluation

The following adversary capabilities are noted in the **persistent data storage extraction** part of the threat model:

- The adversary is able to gain physical access to the IOT devices of the system and extract sensitive data from persistent (i.e., flash) storage, such as secret keys.

Measures/Defense Mechanisms:

- The system implements a PUF module providing an encryption service which leverages a device-unique digital fingerprint as the encryption key. This encryption service is leveraged by the device in order to store an encrypted version of its SSK in persistent storage. To gain access to the decrypted key, the device must request it from the PUF module, providing its own encrypted SSK as input.

- The design and implementation revolves around leveraging the PUF module to never store sensitive information such as the SSK/TSK in persistent storage of the device using plain-text. Sensitive information is always encrypted using the device-unique digital fingerprint reconstructed by the PUF module upon startup.

Simulation:

This attack was not simulated as equipment to simulate such an attack was not readily available. A theoretical scenario can be discussed.

Result:

Considering the mechanisms described, any attempt at extracting sensitive keys from persistent storage in the device would result in the extraction of an

encrypted key, which the adversary does not have any means to decrypt since the device itself does not have access to the digital fingerprint generated by the PUF module.

6 Discussion

This chapter will include discussions on relevant findings from the evaluation chapter and present possible strategies to improve the security and efficiency of the current implementation. The chapter will first discuss possible strategies to address the current vulnerabilities exposed in the security evaluation. Subsequently, possible improvements to the throughput of LDPAs using the data transfer protocol are discussed. Lastly, possible design and implementation changes/additions to protect against a threat model defining a stronger adversary is presented.

6.1 Addressing Current Vulnerabilities

As noted in Subsection 5.3.2, there currently exists vulnerabilities in the implementation. This subsection addresses the existing vulnerabilities in the current implementation and suggests possible solutions to patching them.

6.1.1 Addressing the JSON Issue

As described in subsubsection 4.3.2, message parts sent between the device and the server are wrapped in JSON and sent in the body of the HTTP requests. This design intended to solve a technical issue with sending raw binary data using the `ArduinoHttpClient` library, which proved to be a non-issue after

further investigation.

The problem with this approach is that the wrapping of the message parts require labels, and only the message parts themselves are encrypted. Considering the implementation utilizes HTTP without transport layer security (TLS), these labels are exposed and imply the intention of the encrypted data being transmitted. This provides an adversary with enough information to strategically intercept messages, and more importantly, launch pin-pointed message-modification and DOS attacks to exploit the mutual authentication vulnerability noted in subsubsection 5.3.2.

Possible Solution

Patching this vulnerability would require moving away from the JSON approach entirely, considering the communication should optimally be independent from HTTP. Using TLS would not solve the underlying problem and defeats the purpose of the implementation.

A possible way to send information is to embed the message, its encrypted payload and authentication parameter as raw binary data in the body of the HTTP requests, using the message format illustrated in Figure 6.1. This ensures that the server gets the necessary information with the initial SID, whilst the rest of the message does not expose any intention by labeling specific parts.

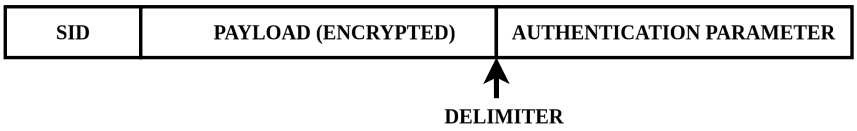

Figure 6.1: Raw binary message format. Delimiter can be be added to support dynamically sized-payloads. Note that the SID and authentication parameters are always 32 bytes.

A small refactoring implementing this was made in the `sendSensorData` function and the appropriate end-point at the server, in order to test the viability of this approach. Due to time constraints, a full revamp of the system was not performed and the temporarily refactored sections still use the JSON approach, but its worth nothing that the strategy is valid.

6.1.2 Addressing the AES256-ECB Issue

As noted in subsubsection 4.3.2, the payload of messages sent between the device and server is encrypted using the `AES256` implementation from the

`Arduino Cryptography Library`. The problem with this implementation is that it uses the ECB mode of encryption by default, which does not include an IV in its encryption process. This results in patterns in the encrypted data, creating a relationship from the plain-text to the cipher-text, susceptible to exploitation by an adversary eavesdropping over a longer period of time.

Possible Solution

Patching this vulnerability could be done by implementing the cipher block chaining (CBC) mode of operation on top of the base ECB block cipher from the `Arduino Cryptography Library`. Note that this would require a slight modification of the raw message format described in Subsection 6.1.1. The new format including the IV is depicted in Figure 6.2.

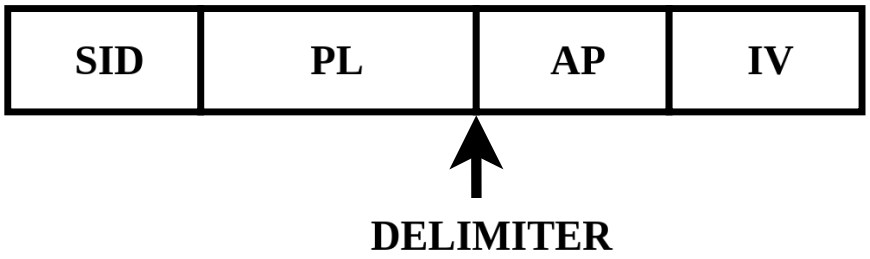

Figure 6.2: Raw binary message format including IV. Note that the format would still only require one delimiter as only the payload might be dynamically sized.

6.1.3 Addressing the Mutual Authentication Issue

As noted in subsubsection 5.3.2 the mutual authentication protocol is vulnerable to message-modification, where the last acknowledgement from the device to the server is either corrupted by modification or lost due to a DOS attack. In the paper by Aman *et al.* [20] which inspired the design, they are aware of this vulnerability in the mutual authentication protocol.

Possible Solution

A possible way to patch this vulnerability is to use the same approach as in [20], where each device and the server stores a list of emergency SIDs. If this were to be implemented using the design incorporated in the thesis project, the `challenge` in the paper would the the equivalent to an encrypted SSK. Note implementing this would also require generating all the different SIDs using the device registration process detailed in Subsection 3.3.2, where each SID

would be generated using a different SSK. To understand how this works, lets play out a scenario where the adversary modifies the last acknowledgement in the protocol:

1. Adversary modifies last acknowledgement.

2. Server verification of the acknowledgement fails, and the server does authenticate the device, nor does it update its registered SID for the device.

3. Device attempts to send data after what it viewed as a successful authentication, but gets a 401 *Not Authorized* response from the server.

4. Device recognizes that the acknowledgement must have been faulty, causing a failed authentication at the server side, and the SIDs are no longer synced.

5. Device uses one of its emergency SID/SSK pairs in its next authentication, which is synced with the server.

6. If the authentication fails again due to the last acknowledgement, another emergency pair can be used.

Note that this solution would require multiple encrypted SSKs as well as SIDs to be stored in the device flash.

6.2 Notes on Efficiency Measurements and Improving Throughput

6.2.1 Small Note on Efficiency Measurements Results

As seen in Table 5.1, the measured section with the largest elapsed time is the `handleSetKeyRequest` function. Although this could be improved, the elapsed time of this function along with the other two key-related functions is considered to be beside the point. The most important measurements is the time elapsed of the `performAuthenticationProtocol` and `sendSensorData` functions, especially the latter considering it is invoked the most by a large margin.

6.2.2 Improving the Throughput of LDPAs

In order to improve the throughput of LDPAs in the system, some small changes could be made in the `sendSendorData` function. Noted in Table 5.1, the function spends 250 milliseconds to complete its procedures, where 130 milliseconds is HTTP communication overhead. The HTTP overhead itself is hard to minimize, although one could use alternative application layer protocol such as message queuing telemetry transport (MQTT) which would perform better in a resource-constrained scenario.

Considering the HTTP overhead accounts for over half the total time elapsed, increasing the batch size of LDPs in the LDPAs could be a viable strategy. This would entail sending more data per request in order to minimize the relative HTTP overhead to the amount of sensor data being sent.

Increasing the batch size of LDPs is a viable strategy. However, at some point, the bottleneck will become the rate of data captures and transfer to the device from the sensor.

6.3 Future Implementations and Extending the Threat Model

While the initial goal of the design and implementation was to safeguard sensitive keys against physical data extraction from persistent storage, the thesis project's work has provided some insights on extending this protection to RAM as well. This section will briefly provides ideas for a new design based on the lessons learned and insight gained from the work in the thesis project. The main idea in the new design is rooted in the knowledge gained throughout the work with the thesis project.

The `LPCXpresso55S69-EVK` development board which implements the current PUF module, allows secure storage of keys within the PUF peripheral of the board [36], specifically in the PUF registers. Keys stored within the peripheral are encrypted using the device-unique digital fingerprint of the board. The board also provides a hardware AES engine and a secure bus from the PUF peripheral to this engine. Essentially this means the development board is capable of decrypting keys stored securely within in its PUF peripheral, before sending them to the AES engine using the secure bus. The AES engine can then encrypt/decrypt data using a secret key received through the bus.

In addition to the above functionalities, the `J7` and `J8` `LPCXpresso55S69-EVK`

expansion connectors (of the expansion header) provide a Mikroe Click module site for MikroElektronika add-on modules [37]. By utilizing this additional interface, the board could be WiFi-enabled by connecting a click board such as the `WiFi 7 Click` [38].

By combining these new insights gained from the thesis work, a suggestion for a future implementation can be presented.

A Future Design:

By WiFi-enabling the `LPCXpresso55S69-EVK` using the `Wifi 7 Click` board, one can eliminate the `Arduino MKR 1010 WiFi` from the design. Originally, the Arduino was only incorporated into the design as the `LPCXpresso` board lacked WiFi capabilities. Figure 6.3 depicts the new design, where the `LPCXpresso55S69-EVK` now acts as a PUF-enabled device with WiFi capabilities. This change in the design includes the following benefits:

- Wires and communication from the device to the PUF module is no longer needed, patching a potential vulnerability and eliminating the overhead it introduced to the system.

- The PUF-enabled device is able to encrypt its own SSKs using its digital fingerprint, and store these in persistent storage.

- The PUF-enabled device provides secure storage of keys in its PUF peripheral and a secure bus to a hardware AES engine. It can now decrypt the required SSK in the PUF peripheral using its digital fingerprint and send them directly to the AES engine utilizing the secure bus. Upon receipt of an SSK, the AES engine can use the SSK for encrypting and decrypting of data. **This means that sensitive keys are never susceptible to physical data extraction methods, as they are never technically stored in regular RAM locations.**

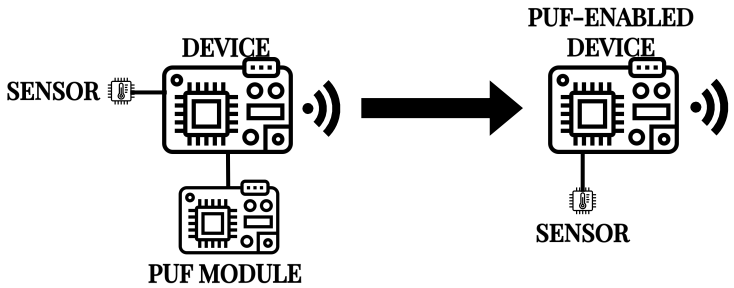

Figure 6.3: PUF-enabled device to safeguard against an adversary capable of extracting sensitive RAM data.

By incorporating the design described above, the new PUF-enabled device can protect its sensitive keys against an extended threat model including an adversary with the capability of extracting sensitive keys present in RAM.

7

Related Work

This chapter discusses related work in the research domain of the thesis project. The chapter explores relevant research employing sophisticated strategies to provide secure data provenance in smart devices or similar devices within the IOT domain. The chapter will describe relatively modern approaches, including labeling schemes, hardware requirements and other relevant information. Note that some of this work exceeds the scope of this thesis and discusses data provenance in the context of geographical location. It is the intent that this information, in combination with the thesis project, could inspire ideas for integrating these sophisticated labeling schemes into the work provided by the thesis project or other systems.

7.1 Related Research on Data Provenance in IoT

7.1.1 Data Provenance and Secure Authentication Using Wireless Channel LQI Measurements and PUFs

Aman *et al.* [20] presents a lightweight protocol for secure data provenance in the IOT using wireless fingerprints. This paper inspired the core thesis project design principle of using a PUF to secure secret keys in devices deployed in compromising environments. Additionally, the authentication and data transfer protocols in the project are adaptions of the protocol presented in this paper.

The main idea presented in this paper is using wireless channel Link Quality Indicator (LQI) measurements for secure data provenance and device authentication using a PUF-based challenge-response authentication protocol. LQI measurements refer to different properties of the wireless signal from an entity. Using these properties which also factors in geographical location, a receiving entity can verify that the sender is legitimate. Data provenance in the context of this paper refers to the tracing of data back to its source, although data labeling is not part of this.

The **threat model** includes an adversary whose capabilities include that of the Man-in-the-middle (MITM), but is also has physical access to devices and is able to extract data by physical means. The protocol in the paper **ensures the integrity of data and authenticity of devices and channels used**.

Note that the strategy proposed in this paper requires a `verifier` implementation and PUF-enabled devices with direct access to the PUF within the device.

7.1.2 Data Provenance and Trusted Authentication by Outsourcing Attribute-Based Signatures and Leveraging Bloom Filters

In a paper by Siddiqui *et al.* [39], a secure mechanism to sign and authenticate provenance messages using Ciphertext-Policy Attribute Based Encryption (CP-ABE) based signatures is proposed. Paraphrasing [39], the technique proposed in the paper leverages *bloom filters* for storage compression and an outsourced Attribute-Based Encryption mechanism to reduce computational demand at the IOT device level. The proposed system uses bloom filters to store provenance information at each node, and uses multiple hash functions operating on the data packets being forwarded in order to set the indexes in each device's bloom filter. The provenance log of a data packet is gathered by querying the bloom filters of the IOT devices in the system, providing the data packet as input to the hash functions determining an index for the packet. If the value in the index is asserted, the device will count this as a bloom filter membership, which indicates that the data packet passed through the queried node.

Note that this mechanism does not support labeling of data, but guarantee the trusted provenance and integrity of authentications and data. The mechanism proposed does not impose any hardware-specific requirements on sensors or microcontrollers.

The **threat model** in the context of the proposed mechanism includes the following assumptions:

- IOT devices are physically vulnerable.
- An adversary can can snoop, alter, reiterate, and infuse invalid data and messages.
- An adversary can impersonate IOT nodes of the system.
- An attacker can modify data sent from an IOT node to the sink and compromise the provenance mechanism.

7.1.3 Data Provenance for IoT using Blockchain Technology

Another strategy is employing blockchain technology to provide secure data provenance, as presented in the paper by Sun *et al.* [40]. The system proposed by the author(s) **guarantees integrity and trustworthiness of provenance data stored in a blockchain** (assumed to be a permissioned blockchain).

Devices register their identities with the blockchain, and data items stored in the blockchain are identified with a UID. Each data item also has an associated provenance log, where provenance information about the data item is stored, such as the identity of the device who stored the item et cetera. Unfortunately there is **no mention of a threat model or a specific definition of an adversary** in this text.

7.1.4 Zero-Watermarking for Data Integrity and Secure Provenance in IoT

The Zero-Watermarking approach by Faraj *et al.* [41] generates zero-watermarks at the IOT device level and embeds data packets being sent with these zero-watermarks, while the actual data remains intact.

Quoting [41], zero-watermarking schemes are defined in the following way:

> In zero-watermarking schemes, watermarks are generated by source node from the extraction important data features of original data without amendment to the data of these features. Different generation functions can be applied in zero-watermarking. The generated watermarks are not embedded in the data payload, but it is invisibly integrated in the data packet and the data remain unmodified.

Sub-watermarks stored in a tamper proof network database are used for regeneration of the zero-watermarks at intermediate locations as well as the

final destination. The re-generation of the zero-watermarks using the sub-watermarks allow the intermediate and the final locations to verify the integrity of the data and query the database for provenance logs of the item(s) in question.

The zero-watermarking approach **guarantees integrity of data and provenance information**. The provenance strategy proposed does not require customized IOT devices, although it does require an implementation of the tamper-resistant database for sub-watermarking storage.

The zero-watermarking provenance strategy is proposed with a slight extension of the MITM **threat model** in mind. In addition to the capabilities of the MITM adversary, the attacker can launch a *database authentication attack*. This refers to an attack where the attacker aims to extract or identity provenance information stored in the network database of the system [41].

8

Concluding Remarks

8.1 Conclusion

In concluding the thesis, revisiting the central questions that have guided the thesis project provides not only a reminder of the core challenges the thesis work aimed to address, but also aids in underscoring the findings and insight gained which could assist future research within the domain. As stated in the introduction, the problem statement that served as the foundation for the thesis work is revisited below:

> *How can the integrity guarantees of sensor data labels be bootstrapped securely at the microcontroller level within smart devices? Furthermore, how can the trustworthiness of these bootstrapped integrity guarantees be ensured, especially in scenarios where devices face threats like physical extraction of sensitive data, such as encryption keys, due to deployment in vulnerable environments? Is it feasible to utilize Physically Unclonable Functions to both bootstrap and sustain the integrity guarantees of sensor data labels generated at the device, considering the aforementioned threats?*

As noted in Section 1.2, the methodology of the work addressing problem statement resembles the design paradigm defined in the intellectual framework for the discipline of computing [7]. The problem statement was addressed through the design and implementation of a prototype system. In accordance with the problem statement, the system implemented leverages PUF technology

at its core to bootstrap integrity guarantees of sensor data and labels at the IOT device level. The design and the implementation of the system is detailed in the thesis, also providing an evaluation of both the security and efficiency of the system. The thesis also provides possible solutions for patching the current vulnerabilities and ideas for future designs.

In conclusion, although vulnerabilities and imperfections exists in the design and implementation, the thesis lays a foundation and verifies the viability of leveraging PUF technology in the problem domain.

8.2 Future work

There are several directions for future work in the context of the thesis, one of which were mentioned in Section 1.4. As the scope of this thesis has been limited using a server as an final destination for sensor data labels, future work in the domain could include integrating the work into a larger scope such as the cloud.

Another direction is to focus on improving the labeling scheme used in the work with this thesis. The current labeling scheme is relatively primitive, and does not enforce any semantic binding between the sensor data and its associated labels. An interesting labeling scheme would be one that allows for separation of sensor data and labels along with manipulating of said data while still enabling enforcement of user access policies.

Bibliography

[1] Thanos G. Stavropoulos et al. "IoT Wearable Sensors and Devices in Elderly Care: A Literature Review." In: *Sensors* 20.10 (2020). ISSN: 1424-8220. DOI: 10.3390/s20102826. URL: https://www.mdpi.com/1424-8220/20/10/2826.

[2] Duarte Dias and João Paulo Silva Cunha. "Wearable Health Devices—Vital Sign Monitoring, Systems and Technologies." In: *Sensors* 18.8 (2018). ISSN: 1424-8220. DOI: 10.3390/s18082414. URL: https://www.mdpi.com/1424-8220/18/8/2414.

[3] Mordor Intelligence. *Wearable Technology Market*. URL: https://www.mordorintelligence.com/industry-reports/wearable-technology-market (visited on 03/19/2024).

[4] European Parliament and Council of the European Union. *Regulation (EU) 2016/679 of the European Parliament and of the Council*. of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). May 4, 2016. URL: https://data.europa.eu/eli/reg/2016/679/oj (visited on 04/13/2023).

[5] Danny S. Guamán et al. "Automated GDPR compliance assessment for cross-border personal data transfers in android applications." In: *Computers Security* 130 (2023), p. 103262. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2023.103262. URL: https://www.sciencedirect.com/science/article/pii/S0167404823001724.

[6] Pedro Machado et al. "A systematic study on the impact of GDPR compliance on Organizations." In: *Proceedings of the XIX Brazilian Symposium on Information Systems*. SBSI '23. , Maceió, Brazil, Association for Computing Machinery, 2023, pp. 435–442. ISBN: 9798400707599. DOI: 10.1145/3592813.3592935. URL: https://doi.org/10.1145/3592813.3592935.

[7] D. E. Comer et al. "Computing as a discipline." In: *Commun. ACM* 32.1 (Jan. 1989), pp. 9–23. ISSN: 0001-0782. DOI: 10.1145/63238.63239. URL: https://doi.org/10.1145/63238.63239.

[8] D. Dolev and A. Yao. "On the security of public key protocols." In: *IEEE Trans. Inf. Theor.* 29.2 (Sept. 2006), pp. 198–208. ISSN: 0018-9448. DOI:

10.1109/TIT.1983.1056650. URL: https://doi.org/10.1109/TIT.1983.1056650.

[9] René Hummen et al. "A Cloud design for user-controlled storage and processing of sensor data." In: *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*. 2012, pp. 232–240. DOI: 10.1109/CloudCom.2012.6427523.

[10] Shenouda. Dawood. *Serial communication protocols and standards : rs232/485, uart/usart, spi, usb, insteon, wi-fi and wimax*. eng. Aalborg: River Publishers, 2020. ISBN: 8770221537.

[11] Douglas Crockford and Chip Morningstar. *Standard ECMA-404 The JSON Data Interchange Syntax*. Dec. 2017. DOI: 10.13140/RG.2.2.28181.14560.

[12] Adobe Systems Incorporated. *JSON Data Set Sample*. https://opensource.adobe.com/Spry/samples/data_region/JSONDataSetSample.html. Accessed: May 6, 2024.

[13] Internet Engineering Task Force. *RFC 4648: The Base16, Base32, and Base64 Data Encodings*. https://datatracker.ietf.org/doc/html/rfc4648. Accessed: May 10 2024.

[14] Christoph Böhm and Maximilian Hofer. "Physical Unclonable Functions in Theory and Practice." In: Jan. 2013, pp. 173–200. ISBN: 978-1-4614-5039-9. DOI: 10.1007/978-1-4614-5040-5_10.

[15] Huansheng Ning et al. "Physical unclonable function: architectures, applications and challenges for dependable security." In: *IET Circuits, Devices & Systems* 14.4 (2020), pp. 407–424. DOI: https://doi.org/10.1049/iet-cds.2019.0175. eprint: https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/iet-cds.2019.0175. URL: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/iet-cds.2019.0175.

[16] *Static Random-Access Memory*. Accessed: April 22, 2024.

[17] Maroua Ahmid, Okba Kazar, and Ezedin Barka. "Internet of Things Overview: Architecture, Technologies, Application, and Challenges." In: *Decision Making and Security Risk Management for IoT Environments*. Ed. by Wadii Boulila et al. Cham: Springer International Publishing, 2024, pp. 1–19. ISBN: 978-3-031-47590-0. DOI: 10.1007/978-3-031-47590-0_1. URL: https://doi.org/10.1007/978-3-031-47590-0_1.

[18] Kourosh Kalantar-zadeh. "Introduction." In: *Sensors: An Introductory Course*. Boston, MA: Springer US, 2013, pp. 1–9. ISBN: 978-1-4614-5052-8. DOI: 10.1007/978-1-4614-5052-8_1. URL: https://doi.org/10.1007/978-1-4614-5052-8_1.

[19] Fernando E. Valdes-Perez and Ramon Pallas-Areny. *Microcontrollers: Fundamentals and Applications with PIC*. 1st. USA: CRC Press, Inc., 2009. ISBN: 1420077678.

[20] Muhammad Naveed Aman, Mohamed Haroon Basheer, and Biplab Sikdar. "A Lightweight Protocol for Secure Data Provenance in the Internet

of Things Using Wireless Fingerprints." In: *IEEE Systems Journal* 15.2 (2021), pp. 2948–2958. DOI: 10.1109/JSYST.2020.3000269.

[21] *LPC55Sxx Usage of the Physically Unclonable Function and Hash*. https://www.nxp.com/docs/en/application-note/AN12324.pdf. Accessed: April 22, 2024.

[22] *LPCXpresso55S69-EVK Development Board*. https://www.nxp.com/design/design-center/software/development-software/mcuxpresso-software-and-tools-/lpcxpresso-boards/lpcxpresso55s69-development-board:LPC55S69-EVK. Accessed: April 22, 2024.

[23] *Arduino MKR WiFi 1010 Development Board*. https://docs.arduino.cc/hardware/mkr-wifi-1010/. Accessed: April 22, 2024.

[24] *Adafruit APDS9960 Proximity, Light, RGB, and Gesture Sensor*. https://www.adafruit.com/product/3595. Accessed: April 22, 2024.

[25] *5-pin JST ESLOV to 4-pin JST SH STEMMA QT / Qwiic Cable - 100mm long*. https://www.adafruit.com/product/4483. Accessed: April 22, 2024.

[26] *Premium Male/Male Jumper Wires (75mm)*. https://www.adafruit.com/product/759. Accessed: April 22, 2024.

[27] *MCUXpresso Integrated Development Environment*. https://www.nxp.com/design/design-center/software/development-software/mcuxpresso-software-and-tools-/mcuxpresso-integrated-development-environment-ide:MCUXpresso-IDE. Accessed: April 24, 2024.

[28] *MCUXpresso Software Development Kit*. https://github.com/nxp-mcuxpresso/mcux-sdk. Accessed: April 24, 2024.

[29] *MCUXpresso Config Tools*. https://www.nxp.com/design/design-center/software/development-software/mcuxpresso-software-and-tools-/mcuxpresso-config-tools-pins-clocks-and-peripherals:MCUXpresso-Config-Tools. Accessed: April 24, 2024.

[30] *Arduino IDE*. https://docs.arduino.cc/software/ide/. Accessed: May 2, 2024.

[31] *WiFiNINA Arduino Library*. https://www.arduino.cc/reference/en/libraries/wifinina/. Accessed: May 2, 2024.

[32] *APDS9960ArduinoLibrary*. https://www.arduino.cc/reference/en/libraries/arduino_apds9960/. Accessed: May 2, 2024.

[33] *Arduino Cryptography Library*. https://rweather.github.io/arduinolibs/crypto.html. Accessed: May 2, 2024.

[34] *ArduinoHttpClient Arduino Library*. https://www.arduino.cc/reference/en/libraries/arduinohttpclient/. Accessed: May 2, 2024.

[35] Wireshark. *Wireshark Documentation*. https://www.wireshark.org/docs/. Accessed: May 11 2024.

[36] *LPCXpresso55S69-EVK Data Sheet*. https://www.nxp.com/docs/en/data-sheet/LPC55S6x.pdf. Accessed: May 11 2024.

[37] *LPCXpresso55S69-EVK User Manual*. https://www.nxp.com/docs/en/user-guide/LPCXpresso55S06UM.pdf. Accessed: May 11 2024.

[38] MikroElektronika. *WiFi 7 Click*. Accessed: May 11 2024.

[39] Muhammad Shoaib Siddiqui, Atiqur Rahman, and Adnan Nadeem. "Secure Data Provenance in IoT Network using Bloom Filters." In: *Procedia Computer Science* 163 (2019). 16th Learning and Technology Conference 2019Artificial Intelligence and Machine Learning: Embedding the Intelligence, pp. 190–197. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2019.12.100. URL: https://www.sciencedirect.com/science/article/pii/S1877050919321398.

[40] Shuang Sun, Huayun Tang, and Rong Du. "A Novel Blockchain-Based IoT Data Provenance Model." In: *2022 2nd International Conference on Computer Science and Blockchain (CCSB)*. 2022, pp. 46–52. DOI: 10.1109/CCSB58128.2022.00015.

[41] Omair Faraj, David Megías, and Joaquin Garcia-Alfaro. *ZIRCON: Zero-watermarking-based approach for data integrity and secure provenance in IoT networks*. 2023. arXiv: 2305.00266 [cs.CR].

[42] OpenAI. *ChatGPT: Conversational AI model*. https://openai.com/chatgpt. 2021.

A

Acknowledging the use of AI in the Thesis Work

In the work presented with this thesis, ChatGPT [42] has been leveraged as a tool to provide inspiration for restructuring of specific sentences that were deemed to long by the author. It has also been used as a verification tool to ensure specific sentences relayed the meaning intended by the author.