University of Zurich[UZH]

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

Detection, Identification and Categorisation of IoT devices on iOS

Aljoscha Schnider
Lachen, Switzerland
Student ID: 17-929-498

Supervisor: Katharina Müller, Dr. Bruno Rodrigues, Prof. Dr. Burkhard Stiller
Date of Submission: February 15, 2023

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

ifi

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

Abstract

The Internet of Things (IoT) has experienced significant growth in recent years, particularly in the area of wireless communication technologies. This thesis focuses on Bluetooth Low Energy (BLE) communication, which has become increasingly important in the IoT ecosystem. Specifically, this research analyses the process of detecting BLE advertisements using an iPhone, and saving the data to an external database. In order to achieve this, a BLE scanner application was developed, capable of storing information found in its vicinity for later analysis.
Due to limiting hardware and the constraints of iOS, it is difficult to gather general rules about advertising data of BLE peripherals. By scanning thousands of devices, it was discovered that a significant number of peripherals either do not share any advertising data or the data cannot be accessed on iOS. The data that is received can be analyzed and utilized to extract device information. The manufacturer can be identified under certain criteria by examining the payload of the advertising data, and it is suggested that device categorisation might also be obtainable.

Acknowledgments

I am extremely grateful for the guidance and knowledge of my primary supervisor, Katharina Müller, who guided me throughout this project. Her expertise and dedication were instrumental in helping me achieve my goals. I would also like to thank Dr. Bruno Rodriguez and Prof. Dr. Burkhard Stiller for giving me valuable feedback. Lastly, I would also like to express my gratitude to my family and friends for their unwavering support and encouragement throughout my thesis.

Contents

Chapter 1

Introduction

1.1 Motivation

Over the last few years, the number of devices communicating via BLE has increased steadily. They allow information to be exchanged quickly and conveniently, but they also entail risks. For example, trackers can be used to follow people without them noticing. The existing defence mechanisms by Apple and Android are sometimes not fast enough to counteract this. For this reason, alternative methods have already been developed to combat the use of BLE trackers. Heinrich et al. have released an Android application that detects such trackers at an early stage and warns the user about their presence. However, someone using an iPhone depends on Apple's internal system solution. In order to get one step closer to a solution for warning users about potential trackers, the possibilities of iOS in the area of BLE are examined more closely. The goal is to find out, how much information can be gathered about advertising BLE devices without connecting to them.

1.2 Description of Work

The goal of this thesis is to answer the following questions concerning BLE on iOS:

- Is it possible to detect devices communicating via BLE and what are the limiting factors?
- Can a BLE device be identified?
- Does the advertising data provide enough information to divide the devices into categories?
- Can the collected data be saved on an external database?

1.3 Thesis Outline

The thesis starts with Chapter 2, Related Work, which presents the inspiration to this thesis in the form of an application on Android, among other papers. It then proceeds to the Theoretical Basis in Chapter 3, where the important parts of Bluetooth, such as the difference between Bluetooth Classic and Bluetooth Low Energy are discussed and an overview of the Bluetooth architecture is given.
Chapter 4 describes the design of the system and what tools and techniques were used. Additionally, it goes over the design of the experiments. The Results will follow in Chapter 5, which presents the final implementation of the system and the results of the experiments. Chapter 6 evaluates the findings before concluding the thesis by answering the questions posed in section 1.2 in Chapter 7.

Chapter 2

Related Work

2.1 Detection of IoT Devices

In July 2020, a paper was released about an iOS application called BTLEmap, which allows users to discover BLE devices advertising in their vicinity. It features, among others, device enumeration, advertisement dissection and a proximity view. The goal of the paper and the resulting application was to improve the understanding of the dangers of BLE advertisements and facilitate access for researchers[1].

2.2 AirGuard

Two years later, the same research group released another application, this time for Android. The application is called AirGuard and it protects its users from getting spied on by devices connected to the "Find My" network. This network is an offline device-finding system created by Apple. Anyone can add a device to the network by using OpenHaystack, which only amplifies the privacy problems [2].
Especially trackers of any kind pose a significant threat as they can be easily hidden to stalk a person [3]. AirGuard counters the misuse of the system by periodically scanning for any BLE devices and reporting suspicious activity. [4]

2.3 Privacy

Privacy and security problems in channel communication were first acknowledged in Bluetooth's version 4.0 and with the release of BLE. Before that, the advertising devices were not anonymised in public channels. Each device has a permanent Media Access Control (MAC) address. In order to prevent stalking, starting with Bluetooth 4.0, devices may use a periodically changing randomized address instead. Becker et al. suggest, that it is possible to track devices even beyond address randomization. [5]

2.4 Database

A study from 2015 shows that a relational database is not as efficient as a non-relational one [6]. To be more precise, the research team compared MongoDB to MySQL, two popular databases. MongoDB was faster in all four basic operations (Insert, Select, Update, Delete), especially when a large amount of data was processed. MongoDB's performance advantage shrinks when the data size is smaller, but does in no way fall off [7].

Chapter 3

Theoretical Basis

3.1 Bluetooth

There are billions of devices all around the world that use the Bluetooth standard. Bluetooth was initially developed in the early 1990s and named after an old Danish King, whose nickname was Bluetooth. The main goal was to develop a technology for short-range data transmission without having to rely on a cable for connectivity.
Nowadays, nearly 30 years after its introduction, there are no signs of stopping the victory march of the Bluetooth standard. In 2021 there were over 4.7 billion devices sold and it is predicted that the number of Bluetooth-enabled appliances will only increase further [8]. Over the years, the Bluetooth standard was continuously updated and improved by the Bluetooth Special Interest Group (Bluetooth SIG). The Bluetooth SIG was first founded in 1998 by five companies, namely the Nokia, International Business Machines (IBM), Intel, Toshiba and Ericsson Corporations. Nowadays, the organization acts as an overseer of the Bluetooth standard's development. They license Bluetooth technology to manufacturers but do not produce any devices themselves.
At this point, an important distinction needs to be made. In 2010 Bluetooth 4.0 was introduced and with it, Bluetooth Low Energy which is also referred to as Bluetooth Smart. Every version before that was retrospectively called Bluetooth Classic. This thesis focuses on BLE, but Bluetooth Classic and BLE have a lot in common [8]. This chapter will heavily rely on Robin Heydon's book called "Bluetooth Low Energy: The Developer's Handbook" [9].

Overview and Design Goals

First introduced in 1998, Bluetooth Classic acts as a short-range connectivity solution for devices. It can transmit up to 3 Mbps of data but consumes a lot of power by doing so. It can uphold a connection over a long period of time and is therefore mostly used in devices which need to transmit data continuously, like wireless speakers or headphones.

Just like Bluetooth Classic, BLE also uses the 2.4 GHz Industrial/Science/Medical (ISM) band, which is legal everywhere and can be used without a license. But in stark contrast

to Bluetooth Classic, BLE strives to use as little power as possible. However, this also means that not only do less high data rates have to be accepted, but the connection should also be kept as short as possible. Otherwise, too much energy would be consumed too quickly, which leads to higher costs in production as companies would need to implement more powerful and thus increasingly expensive batteries.
Cheap production is at the heart of the philosophy around BLE in particular. This can be determined by the following three points in the design concept: The usage of the ISM band, how IP licensing is handled and the low power needed to run BLE. The ISM band, that is used, is available worldwide and license-free. This significantly reduces the cost of a BLE device, as no licensed spectrum is needed. However, there are also significant disadvantages, after all, interruptions can easily occur, as other technologies also use the same spectrum. Moreover, the radio energy is easily absorbed by everything, even making it a "terrible place to design and use a wireless technology" according to Heydon.
Nevertheless, the financial advantages clearly outweigh the functional disadvantages. Another financial advantage is, how the licenses are distributed. The Bluetooth SIG is responsible for issuing the licences to use Bluetooth (Classic and BLE) and is known to have an excellent reputation. This is partly because they issue the licences at a reasonably low price. That is the reason why BLE (and Bluetooth in general) has very low licensing costs per device.
Lastly, BLE does not consume much power and it is quite easy to save budget on the hardware. Less power equals fewer materials used and therefore less money spent on the device, which in turn makes it cheaper for the customer as well. Ultimately, the goal is to use the button-cell battery, as it is the cheapest, smallest and most readily available type of battery on the market.
Bluetooth Low Energy was specifically designed for situations, where only a limited amount of data needs to be transmitted at once but the devices should get by with little energy over a long period of time.

3.2 Architecture of BLE

To remain faithful to the philosophy of BLE, the architecture needs to be kept as simple as possible in order to keep the costs down. It consists of three main parts: The controller, the host and the application. This chapter will look into each of them individually. Heydon's "Bluetooth Low Energy: The Developer's Handbook" was used as a source [9]. Figure 3.1 is an overview of each section.

3.2.1 Controller

The controller is the physical part that allows the transmitting and receiving of packets.

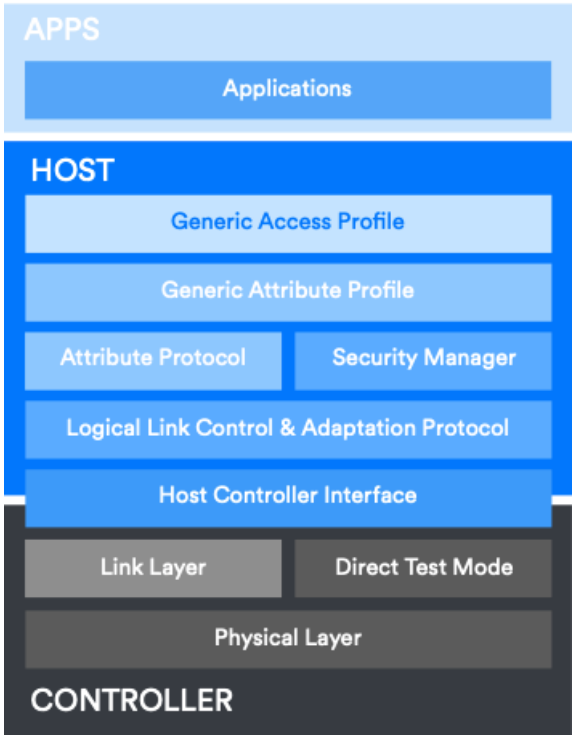


Figure 3.1: The Bluetooth Low Energy Stack Overview

The Physical Layer

The Physical Layer is responsible for the translation of digital symbols over the air. It communicates through radio waves, which carry the information using the Gaussian Frequency Shift Keying modulation scheme. GFSK is a type of frequency modulation whereby the bits are represented by the variation of the carrier waves' frequency instead of its amplitude [10]. It decreases the frequency whenever a 0 needs to be transmitted and increases it in the other case. Additionally, Gaussian filtering is applied as well, resulting in the GFSK [11]. As discussed before, the benefit of using this scheme is once again its low cost, but it is not optimal for usage in noisy environments [12].

The Direct Test Mode

The Direct Test Mode is an approach to make the physical layer testable. It allows a user to communicate directly with the physical layer and test its transmission and reception capabilities.

The Link Layer

The Link Layer is responsible for the most important part of the Bluetooth Low Energy architecture: Advertising its state and handling connections. As the connecting part is not featured in this thesis, this section will not go into detail on parts responsible for

that aspect but it is nevertheless a huge part of the Link Layer. Advertising is of greater interest in the scope of this thesis.
To lower the power and therefore the cost while still being robust, the Link Layer uses three advertising channels. On those channels, advertising data is sent in the form of small packets. But before diving deeper into packet structure, the Link Layer state machine needs to be introduced. The Link Layer state machine defines five possible states it can be in, although only one can be active at any given time.
The Standby state is the inactive state, the device is neither transmitting nor receiving packets. This state is reachable from any other state and it is the starting point for any device. In the Advertising state, the Link Layer transmits advertising packets. It is crucial if a device wants to be discoverable, connectable, or broadcast data. If the device wants to use this state, a transmitter must be implemented, a receiver is optional. Once the Advertising state is active, it is possible to move to the Connection state, if a connect request was sent, or to the Standby state, by simply stopping to advertise data. There are four different types of advertising:

- General Advertising (connectable undirected)

  When a device advertises generally, it can be scanned and/or be connected to by any device if it receives a connect request.

- Direct Advertising (connectable directed)

  Direct Advertising is used whenever a device needs to connect to another already-known device quickly and without delay. The direct advertising packets consist of the advertiser's and the initiator's addresses. In this case, the initiator is the device to which the advertiser wants to connect to.

- Nonconnectable Advertising (nonconnectable undirected)

  This type is used by devices, which do not want to connect to any other device. On top of that, they can not be scanned for any additional data. This is mostly used by broadcasters as a receiver is not necessary.

- Discoverable Advertising (scannable undirected)

  Discoverable Advertising is used to allow devices to scan its data, but not to connect to it. So, the devices which use this type of advertising, broadcast data and return a scan response if asked for it.

The third state, the Scanning state, consists of two individual substates: Active and passive scanning. Both states have in common that advertising packets can be received, but active scanning additionally sends scan requests to the advertising devices. This can lead to more data in form of a scan response. Devices which only support passive scanning, do not need to have a transmitter. That is only necessary if it should support active scanning as well, as a scan request needs to be sent.
If the scanning process is stopped, the device returns to the Standby state automatically. That is the only state it can move to. The device moves to the Initiating state whenever it wants to initiate a connection to another device. The only state, that it can move to, is back to the Standby state by stopping initiating a connection. Lastly, in the Connection

state, data can be exchanged between the two connected devices. It is the only state, where the three advertising channels are not used anymore, instead, it relies on 37 data channels.

Moving on to the packet structure, which can be seen in figure 3.2. The first byte is the preamble and is used by the receiver for synchronisation and to set its automatic gain control. For advertising packets, its content in binary form is always ”10101010”. The following four bytes represent the access address. For the three advertising channels, this address is constant: 0x8E89BED6. Therefore, both the preamble and the access address are fixed values for advertising packets. The header consists of some bits that indicate if the advertiser's address is public or random as well as the advertising packet type.

There are seven different advertising packet types and each one has a different payload format:

- ADV_IND = General advertising indication
- ADV_DIRECT_IND = Direct connection indication
- ADV_NONCONN_IND = Nonconnectable indication
- ADV_SCAN_IND = Scannable indication
- SCAN_REQ = Active scanning request
- SCAN_RSP = Active scanning response
- CONNECT_IND = Connection indication

Those correspond to the types of advertising, the active scanning process or the Initiating state, respectively. The next 8 bits are the length of the payload, the actual data which is transmitted. The payload itself varies greatly, from advertising data about the device itself to service data or scan response data (section 3.2.2). But the advertiser's address is always included. This address can either be public, random static, random non-resolvable or random solvable. Simply sharing the globally unique MAC address of the device is the public option. Random static means that the address is randomly generated but should stay the same for as long as possible. Although it could be renewed whenever the device was rebooted.
In contrast, a random non-resolvable address can always be generated anew. Random resolvable addresses work the same but add some functionality, which allows a central device to resolve the address if they have already connected to the advertising device before. The Bluetooth Specification encourages manufacturers to change the random addresses every 15 minutes [13]. The last part of the packet structure is the Cyclic Redundancy Check (CRC) which is responsible for detecting possible errors in the received packets.

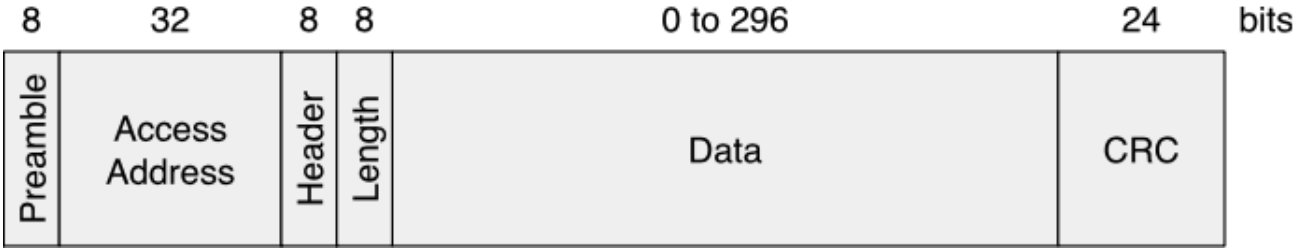


Figure 3.2: Packet structure of the Link Layer

The Host/Controller Interface

The HCI is responsible for the internal communication between the controller and the host. The host sends commands and data to the controller whereas the controller sends events and data to the host. As the HCI is implemented in both the controller and the host, the part located in the controller is called the lower-host controller interface and the part located in the host is the upper-host controller interface.

3.2.2 Host

The host is built on top of the upper-HCI and contains the following protocols and profiles. This section will not go into much detail on every one of them, as not all are important for this thesis.

Logical Link Control and Adaptation Protocol

The Logical Link Control and Adaptation Protocol (L2CAP) is responsible for the adaptation between the higher and lower layers of the Bluetooth stack. It segments and reassembles data packets to enable communication as the lower layers can not handle the same capacity as the upper layers. Additionally, the L2CAP does the multiplexing part of BLE.

The Security Manager Protocol

This protocol allows BLE to recognize devices. As this is only applicable after connecting to a peripheral, it is not important for this thesis.

The Attribute Protocol

The Attribute Protocol handles incoming requests among other operations from a connected device. It enables communication whenever the connecting process was successful.

The Generic Attribute Profile

The Generic Attribute Profile (GATT) procedures define how data can be discovered and used afterwards. It uses the Attribute Protocol as its transport protocol to transport data between devices.

The Generic Access Profile

This profile defines how a device can discover other devices and how to connect afterwards. Additionally, it states how advertising data is structured. First, the four GAP roles need to be explained as they enforce restrictions on how devices can communicate with each other.

- Broadcaster

  A broadcasting device only transmits data, but does not receive them. It can not be connected to nor asked for a scan response and therefore does not need a receiver.

- Observer

  An observer device is the counterpart to the broadcaster, as it just scans for any advertising broadcasters in its vicinity. Those devices do need a receiver but not necessarily a transmitter.

- Peripheral

  A peripheral device advertises as well, just like a broadcaster, but does allow connections. Therefore, it needs both a receiver and a transmitter.

- Central

  Central devices initiate connections to peripherals. Similar to peripherals, they also rely on both the receiver and transmitter.

It is important to know that a device may support multiple GAP roles. A good example is the smartphone, as it can act both as a peripheral and central. Apart from the different roles, there are two other important basic concepts within GAP: The modes and procedures. Modes are states in which a device can be in for a longer time whereas procedures are sequences of actions within a finite time interval. This section only mentions the most important ones as not all need to be addressed. Note that all modes concerning discoverability and connectability are only applicable to devices which are in the peripheral role.

- Broadcast mode

  In this mode, the device is simply broadcasting data. This is the only mode a device taking on the broadcaster role can be in.

- Nondiscoverable Mode

  Being in the nondiscoverable mode means that it can not be detected by any scanning device.

- Limited-discoverable Mode

  A device is only allowed to use the limited-discoverable mode for about 30 seconds and is often utilised when a user has recently directly interacted with it by, for example, pushing a button. This is used to make a certain device stand out amongst the other advertising devices in its vicinity and oftentimes uses a faster advertising interval than general-discoverable devices.

- General-discoverable Mode

  This mode is mostly the same as the limited-discoverable mode but differs in two aspects: A device can stay in that mode for an unlimited time and the recommended advertising interval is slower.

- Nonconnectable Mode

  The nonconnectable mode can be used if a device should not connect to another device.

- Directed-connectable Mode

  This mode is used whenever a peripheral wants to connect to a specific central quickly. It can only be used for a short amount of time. In this mode, the peripheral is not discoverable by other devices.

- Undirected-connectable Mode

  This is the standard mode for allowing a peripheral to connect to any other device. It can stay in this mode for an unlimited amount of time, similar to the general-discoverable mode.

- Discovery Procedures

  If a central starts scanning for any advertising devices, it uses one of two possible discovery procedures: Limited-discovery or general-discovery procedure. The difference is that the limited-discovery procedure only detects data which was transmitted in an advertising packet from a device in limited-discoverable mode. General-discoverable peripherals' advertising packets are ignored. The general-discovery procedure on the other hand discovers all advertising packets.

The Generic Access Profile also defines how advertising data is structured. It consists of a sequence of advertising data structures whereby each structure has a length field, an advertising data type and lastly an actual value.
The length field specifies the number of bytes the structure consists of, excluding the amount it uses itself. There are many different advertising data types, this thesis focuses on the local name, service UUIDs, service data, TX power level and manufacturer data.
The local name advertising data type exposes the local name of the advertiser, either in complete form if there is enough space, or shortened. The incompleteness is also present

in service advertising data types. There are four in total: A complete and a partial list for 16-bit and 128-bit service UUIDs (Universally Unique Identifier) respectively. Those services are collections of data and associated behavior which are exposed by the advertiser. A good example is a light bulb, one service it could provide would be the information if the light is turned on or not. The Bluetooth SIG has predefined certain services which are often used but it is also possible to create new services. Companies, who are members of Bluetooth SIG, can request such service UUIDs, which are then listed in the Assigned Numbers document [14]. The list only refers to the company that owns the respective 16-bit UUID, but not to the name of the service.
A service can also hold data, which is almost completely defined by the company that produced the advertising device. The first two bytes are a 16-bit service identifier, which is again accessible via the Bluetooth SIG document [14]. The power level used to transmit the advertising packet is exposed through the TX power level advertising data type. Lastly, the manufacturer-specific data stands for data the manufacturer declared themselves. Similar to the service data, the first two bytes of the manufacturer-specific data correspond to a company identifier in the Assigned Numbers document.

3.2.3 Application

The final layer sits on top of the GAP and is that part of the stack, which directly interacts with the user. It contains the UI, application logic and overall architecture. The application can be split into two groups: The design of applications running on a central device and the design of applications running on a peripheral device.

3.3 IoT Devices

The Internet of Things (IoT) is the term used to describe the network of physical objects (things) that are equipped with sensors, software, and other technology to connect them to other devices and systems over the Internet so that data can be exchanged between these objects. Popular examples of such devices include Smart Home devices like smart thermostats, security systems and smart lights or wearables such as smartwatches, fitness trackers and heart rate monitors. But they are also used in the economy through so-called industrial IoT devices, for example in sensors for monitoring temperature, humidity and pressure in manufacturing and industrial environments. IoT devices are transforming conventional products into connected ones, offering innovative and more effective ways to automate, monitor, and control various aspects of our lives and the environment.

Chapter 4

Methods

In order to make the evaluation process of the research questions even possible, an iPhone application, as well as a back-end and an external database were created. They should be able to communicate over a local area network without issues.
This chapter serves as an overview of the application itself and everything that was used to ensure its functionality. The application needs to run on an iPhone, so the programming language Swift will be used, as it is the most modern language in which iPhone operating system (iOS) applications can be written. Additionally, Swift is always being developed further and therefore Apple adds more language enhancements to Swift than to Objective-C, the language used before the introduction of Swift in 2014. It is suggested, that Swift is not only more efficient than its predecessor but also less complex in its structure [15].
Figure 4.1 serves as an overview of how the whole system works. The application, which runs on an iPhone, collects data from various IoT devices in its vicinity, before giving the user the opportunity to save it on an external database. This is done by sending the data to the back-end via HTTP, which runs on MacOS and then sending the data to an external database hosted by a Raspberry Pi.

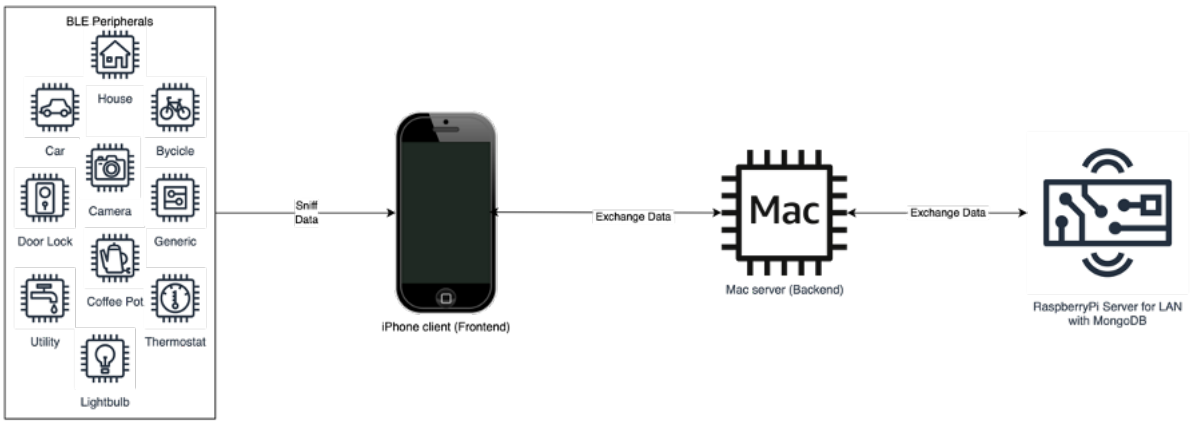


Figure 4.1: System Layout

4.1 Front-end: SwiftUI

The front-end is written using SwiftUI, a user interface framework developed by Apple for intuitively creating UIs for iOS and other Apple platforms. It was first introduced in 2019 as a more modern approach for building UIs, replacing the previous framework UIKit [16].
The application should have two separate screens: The main screen which features a scanner and a second one which displays the entries that were already saved. A tabbar is used to switch back and forth between the two screens. The main screen should feature buttons to start and stop scanning for advertising devices. Both should reflect their respective entries in a list. As each entry should represent information about a scanned device, it should be possible to get further into detail by interacting with a row. This should lead to a view which features all vital insights as well as an option to either save or delete the entry, depending on the screen.

4.1.1 BLE on iOS

BLE on an iPhone is used via a framework which is called Core Bluetooth, is provided and documented by Apple [17]. More specifically, it is an abstraction of the BLE stack that makes it easier for the developer to work with. While using the application, the iPhone takes on the role of the central device, while the surrounding IoT devices serve as peripherals. The app therefore scans for any nearby devices that are advertising and shows the collected information to the user.
In order to start the scanning process, a CBCentralManager object first has to be created. This object handles communication with any remote peripheral device. In this app, scanning and reporting the discoveries is all it has to do but to achieve that the corresponding delegate object has to be set first. The specified object needs to conform to the CBCentralManagerDelegate protocol, which means the centralManagerDidUpdateState() method has to be implemented. This method is called whenever the state of the central manager changes.
There are multiple possible states the centralManager can be in:

- .poweredOff

  Bluetooth is off, which occurs mostly when the user did not turn it on in the settings or in the control center.

- .poweredOn

  Bluetooth is on and can be used, as the required prerequisites were met. The ideal case to start using the centralManager object.

- .resetting

  There was a connection issue concerning Bluetooth and it is trying to reconnect. This can also happen if the user turns Bluetooth off and on again quickly.

- .unauthorized

  This state indicates that the app does not have permission to use Bluetooth. Needs to be made available in the settings.

- .unknown

  The current state is not known, could happen if Core Bluetooth was not yet initialized.

- .unsupported

  The device can not use Bluetooth as it is not equipped with it. Nowadays most iPhones have Bluetooth services but it can occur if the iPhone's generation is older than 4s. Always comes up if the app is tested on the simulator of XCode as it is not possible to use any Bluetooth functionality with it.

To start the scanning process, the centralManager first needs to be activated by turning Bluetooth on in the settings and allowing the app to use it. The state updates to the desired .powerOn and now all methods of CBCentralManager can be used. The most important ones will be analysed in the next paragraphs.

```
func scanForPeripherals(withServices serviceUUIDs: [CBUUID]?,
options: [String : Any]? = nil)
```

After turning the centralManager object on, the scanForPeripherals() method can be called which scans for any advertising peripherals in the vicinity of the device.

```
func centralManager(
_ central: CBCentralManager,
didDiscover peripheral: CBPeripheral,
advertisementData: [String : Any],
rssi RSSI: NSNumber
)
```

Whenever a peripheral was discovered during the scanning process, this delegate method is called automatically. It is optional to implement but is vital in this application. There is no way of obtaining the raw data received from the peripheral. Inside the method, the discovered CBPeripheral object can be used to, for example, start a pairing process. The method also gives access to the RSSI value, which can be used to approximately calculate how far away the peripheral might be.
Most importantly it provides the advertisement data of the peripheral if there is any. The advertisement data on iOS consists of a dictionary containing different key/value pairs. It is not guaranteed to have all of the following, or even any to be precise. Many of them correlate to the advertising data types in section 3.2.2.

- CBAdvertisementDataLocalNameKey

  This key returns the local name of the peripheral as a String.

- CBAdvertisementDataManufacturerDataKey

  The value of this key is the manufacturer data the peripheral might have, the first two bytes represent the Bluetooth SIG company identifier in little endian order. If it starts with Apple's company identifier (0x004C), it is immediately stripped away by iOS [1].

- CBAdvertisementDataServiceDataKey

  This key leads to a dictionary with CBService UUIDs as keys with associated service-specific data with type NSData.

- CBAdvertisementDataServiceUUIDsKey

  The value associated with this key is an array of service UUIDs.

- CBAdvertisementDataOverflowServiceUUIDsKey

  Similar to CBAdvertisementDataServiceUUIDsKey, this key is associated with an array of CBService UUIDs, but those did not fit into the main advertisement.

- CBAdvertisementDataTxPowerLevelKey

  This key leads to an NSNumber representing the transmitting power of the peripheral. Useful to calculate the path loss when taking the RSSI into account.

- CBAdvertisementDataIsConnectable

  Associated with this key is either 0 or 1, indicating if it is currently possible to connect to the device. This is determined by the advertising type: Both general or direct advertising lead to 1 as a result, any other type will return 0 and are not connectable (section 3.2.1).

- CBAdvertisementDataSolicitedServiceUUIDsKey

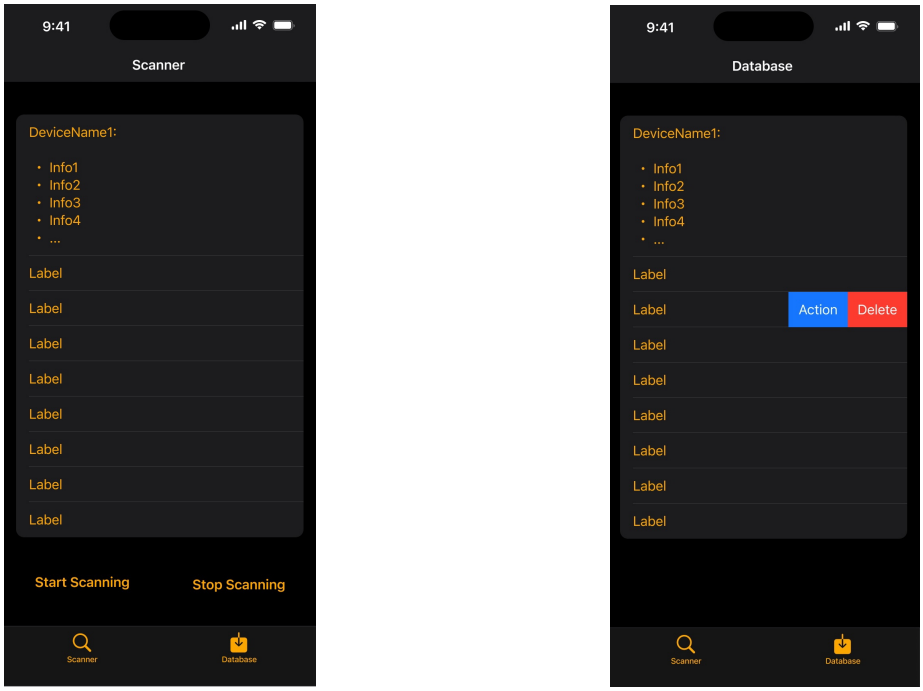
  The associated value is an array of UUIDs which the peripheral wants the central to support.

Apart from the data gained by inspecting the advertisement data, the peripheral object can also be used. Its identifier property can be utilised to recognise peripherals, although its data type is a UUID which is created by iOS with the peripheral's (public or random) address as input among others. That means that it varies between iOS devices, even though both get the same advertising packet from the same peripheral.

4.1.2 Design Mockup

The application is expected to have the following features:

- Should be supported from iOS 14 onwards.
- A BLE scanner, which is able to scan for advertising packets and show them to the user.



(a) Scanner (b) Database

Figure 4.2: UI Mockups

- The option to start and stop the scanner.
- A possibility to inspect a specific device thoroughly and to save it inside of the database.
- An overview of all peripherals that are currently in the database and the opportunity to delete them.

To visualise these features, a mockup was created, which shows what the initial idea looked like. The scanner in figure 4.2a was designed to be as intuitive as possible, only featuring two buttons for the most essential tasks. The main attraction is the list, featuring all devices that were scanned and their respective data visualised in a sub-list. The database screen should look quite similar, just without the buttons. Each row in both screens should have its own slide function, which can be used to delete or add the entry. This feature can be seen in figure 4.2b, on the right side of the third row.

4.2 Server/Back-end: Swift

The server runs on a Mac, as it is also an XCode project written in Swift. This was done specifically so that both the application and the server are written in the same language, making it easier to use and extend in the future. Additionally, Vapor was used for this project. It is an open-source web framework used as a back-end for the application. It has official drivers for MongoDB, making it especially useful as it is compatible with the

instance that runs on the Raspberry Pi. At the moment Vapor is the main high-level server-side Swift framework on the market [18]. It is built on top of SwiftNIO which is an asynchronous networking library and it handles all HTTP communications. So this is the basis on which Vapor takes care of the higher-level aspects of a server.

4.3 External Database: Raspberry Pi

The Raspberry Pi is a small-scale but fully functional single-board computer. In this thesis, it is used as an inexpensive way to make sure the database is not only detached but also scalable as well as maintainable [19]. If a database is capable of running on hardware with such limited resources, it shows that it does not rely on much computing power and could theoretically handle the persistence of much more data if properly scalled-up. Ubuntu 20.04 is used as the operating system, as the newer ones do not yet work properly with MongoDB. The exact installation steps can be found in the appendix. Apart from the operating system, Docker as well as MongoDB were additionally installed. Those will be explored further in the next two sections.

4.3.1 Docker

Docker is an open-source platform for developers to create, run and ship applications. It allows the user to deploy applications on different hardware without having to rely on the specific operating system of the host computer. This ensures that the application works the same way on every platform and therefore facilitates the delivery and distribution of applications. That is done with the help of so-called containers, which are rather independent of the host and can be equipped with all necessities to run any given application. A container is lightweight and does not use many resources of the host computer [20].

4.3.2 MongoDB

MongoDB is a popular, open-source non-relational database which is used for storing the collected data. It was specifically chosen because of its scalability and speed as seen in section 2.4. MongoDB stores data in flexible binary JSON (BSON) documents, which supports more data types than JSON and is widely used with the chosen database. Although MongoDB stores data in BSON, as a developer you still work with the known JSON format, as the conversion happens internally.
MongoDB is designed to scale horizontally, which means that it can handle increasing amounts of data by adding more servers to the system. This is in stark contrast to traditional relational databases, which often have to scale vertically by increasing the power of a single server. Horizontal scalability allows MongoDB to handle very large amounts of data without compromising performance. It does that by making use of sharding, a technique where the data is divided into smaller sets and spread to the aforementioned servers.

Additionally, MongoDB stores most of its data directly into the RAM, allowing for quicker saving as well as retrieving of information. Another key feature of MongoDB is its support for rich data models. The flexible document structure allows for a wide variety of data types, including arrays and nested documents [21].

Structure

MongoDB organizes its data in layers. The top layer is a database, which consists of one or more collections. A collection holds one or more documents whereas a document in turn holds the actual data. It is structured into field/value pairs. The field names are Strings, but the values can be any of the BSON data types. It is even possible to embed another document. An important rule is that there is always a field named "_id". Its value must be unique in the corresponding collection, as it serves as the primary key.

BSON

BSON is a data format which supports more types than JSON. For example, it adds dates or ObjectID, which is a data type that allows the database to easily distinguish one document from the other as it is used as the primary key of MongoDB. Additionally, it was created specifically to improve MongoDB's efficiency and for that reason it was chosen for this project.

4.4 Experiment Design

To evaluate the system, the different features of the application are tested. Regarding the scanner, two experiments were set up. Additionally, the connection between the application, the back-end and the database was tested with a unit test using Apple's XCTest framework.

4.4.1 Controlled Environment

The first experiment is carried out under controlled conditions. Only one peripheral advertises at a time and can therefore be identified regardless of the amount of information in the advertising data scanned by the application. The data collected is analysed and an attempt is made to reconcile it with what is known about the device, such as the brand, manufacturer or device type. The goal is to find general solutions for detecting, identifying and categorising devices communicating over BLE. Only a handful of devices were tested due to the limiting factors regarding the available hardware.

4.4.2 Public Environment

The second experiment is carried out in a public environment with the goal to retrieve as many advertising devices as possible. The collected data cannot be manually traced back to the respective devices, therefore it is more of a test of how much information can be retrieved by scanning.

4.4.3 Database

The database part is tested in a LAN by starting the server and the Raspberry Pi. As the front-end is not part of the test, UI updates do not matter and the main thread does not need to be used. The savedPeripherals in fetchPeripherals() inside of the Model-ListViewModel.swift file can therefore be allocated without it being on the main thread. Additionally, the call to fetchPeripherals() in deletePeripherals() occurring in the same file can be left out as well.

The database is empty upon starting the test. The application then dumps 1000 entries to the database before checking with the get request if the correct amount was saved. Afterwards, each device gets deleted again. In the end, there should be as many peripherals in the database as before starting the test.

Chapter 5

Results

5.1 Front-end

This chapter analyses the user interface of the application and explains the final implementation choices bases on the design mockups in section 4.1.2. It is a deep dive into each file and goes over the essential code of the project.

5.1.1 BAArbeitApp.swift

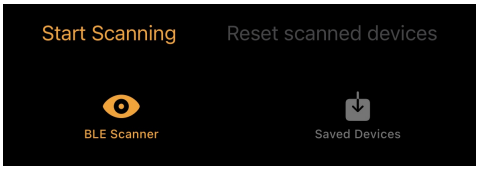
This is the entry point of the application. It presents the ContentView file and initializes a ModelListViewModel instance to be used throughout the whole application. The object facilitates the updating process of views, as whenever it gets updated, the views where the viewModel was used update as well.

5.1.2 AppDelegate.swift

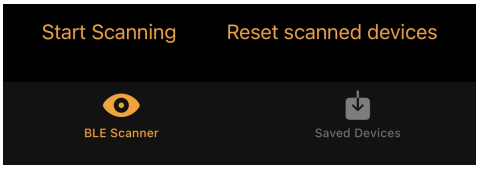
Here is the only instance where UIKit is used in the whole application. This is done in order to set the tabbar visually apart from the views. It appears less dark than the rest, which helps distinguish it as seen in 5.1. As SwiftUI does not support the same color as prior iOS versions.

5.1.3 Utilities.swift

This file is responsible for communication with the back-end. It is taken from an example shared on the official MongoDB website with minor changes such as updating the used model as well as the baseURL [22].



(a) Bluetooth off



(b) Filter Options

Figure 5.1: Tabbar

5.1.4 BLEManager.swift

The BLEmanager class is the implementation of the needed Bluetooth components discussed in the section 4.1.1. The CBCentralManager object is created upon initializing the class, important to notice that there can only be one instance of BLEManager at once and is therefore a Singleton class. The intention behind it is that there are no complications with multiple CBCentralManager objects. The most important part of the class is the didDiscover function. As explained in section 4.1.1, it is called whenever the application finds an advertising peripheral. The function can be split into different parts.

```
for device in viewModel.scannedPeripherals {
        if device.identifier.uuidString == peripheral.
          ↪ identifier.uuidString {
            return
        }
  }
```

Firstly it is checked if the peripheral was already scanned beforehand. This is done by comparing the UUIDs of all peripherals saved by the ViewModel to the discovered peripheral. If they match up, the device was already discovered before and can be ignored.

```
if let name = peripheral.name {
        peripheralName = name
  }
```

Next, it looks for any name the peripheral might have. The advertisementData dictionary could hold a value for the name as well but as the application serves as a scanner, the name property of the peripheral is the same. Only after connecting it might change but that is out of scope for this thesis. [23]

5.1.5 ModelListViewModel.swift

During the whole front-end, an architectural design pattern called Model View View-Model (MVVM) is used. In its essence, it is about how the code is structured and this particular approach focuses on separating logic from the actual UI. The ModelListView-Model class is the logic behind many views used in the application. It conforms to the ObservableObject protocol, which enables any property with the wrapper @Published to trigger UI updates. The variables ”scannedPeripherals” and ”savedPeripherals, which are arrays of BLEDevices, can therefore be changed and the UI reacts accordingly.

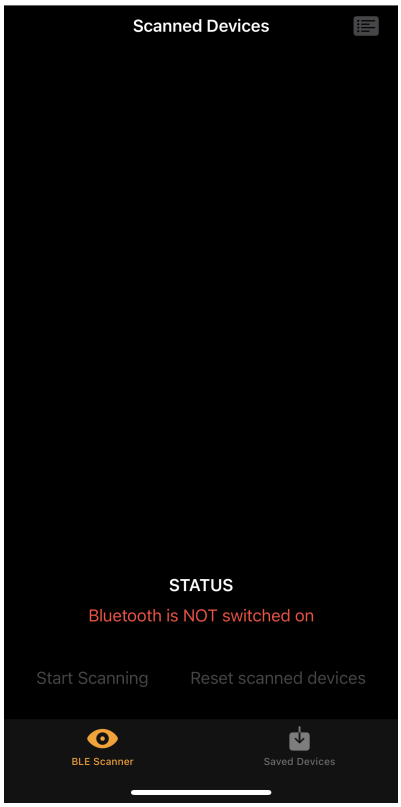
The ModelListViewModel is responsible for interacting with the back-end. It fetches, adds and deletes the devices as well as changes them if necessary. This class does all the heavy work whereas the views only need to know how to present the data and how to call the corresponding functions of the ModelListViewModel class.

5.1.6 Enums

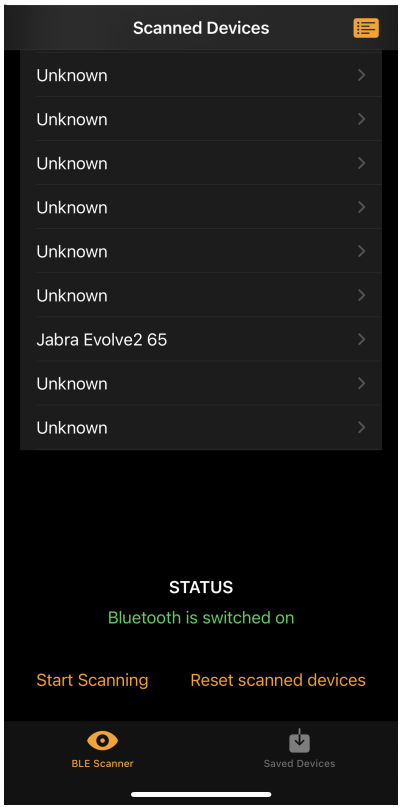
The Enums folders consist of different files mostly used for type safety. DeviceType.swift contains different types of devices of which the peripheral might be part of. BLEManu-facturers behaves similarly as it contains a finite list of manufacturers of BLE devices. It additionally includes a function to assign a manufacturer to a given company Identifier as well as a member to service UUID.

5.1.7 MainView.swift

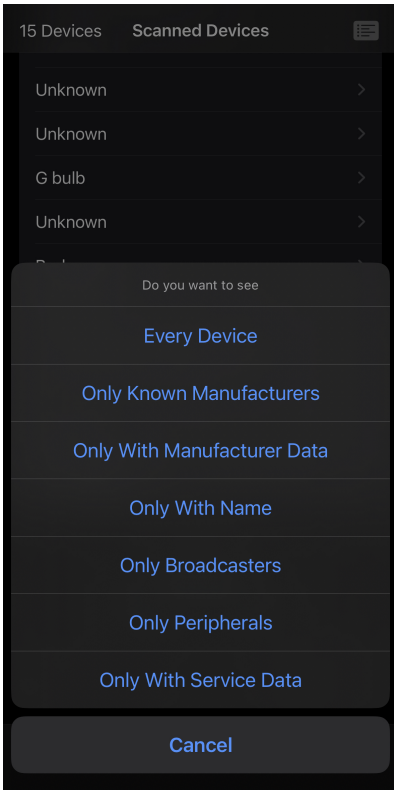
This view is the first a user gets to upon starting the application as it presents a tab view consisting of the ScannerView and the SavedView. The ScannerView is declared in this file, presented instantly and the result can be seen in figure 5.2a.
Bluetooth is not activated, that could be because the application does not have permission (yet) or Bluetooth was turned off in the iOS settings. Figure 5.2b shows how the applica-tion looks like when Bluetooth is turned on and the application has already scanned a few devices. It differs from the design mockup, especially how the list is presented. If every entry had displayed all the information directly, the view would quickly have become very unclear and confusing. That is why the decision was made to put the gathered advertising data in another view, which is accessible by tapping on an individual row (section 5.1.9). The localname serves as the title for each entry. Instead of having the start and stop scanning button both simultaneously on the screen, they alternate depending on the state of the scanning process. An additional button has been added to provide more clarity by resetting the list of devices and start over.
Another useful feature was introduced in the form of a device filter on the right of the navigationbar. This enabled the user to choose between seeing all discovered peripherals or only those that fulfil a certain criterion. The possibilities can be seen in figure 5.2c. That was implemented to facilitate searching for a specific group of devices without having to constantly reset the already discovered ones or look through each entry individually. Lastly, the Bluetooth status was added.



(a) Bluetooth off



(b) Bluetooth on, some devices scanned



(c) Filter Options

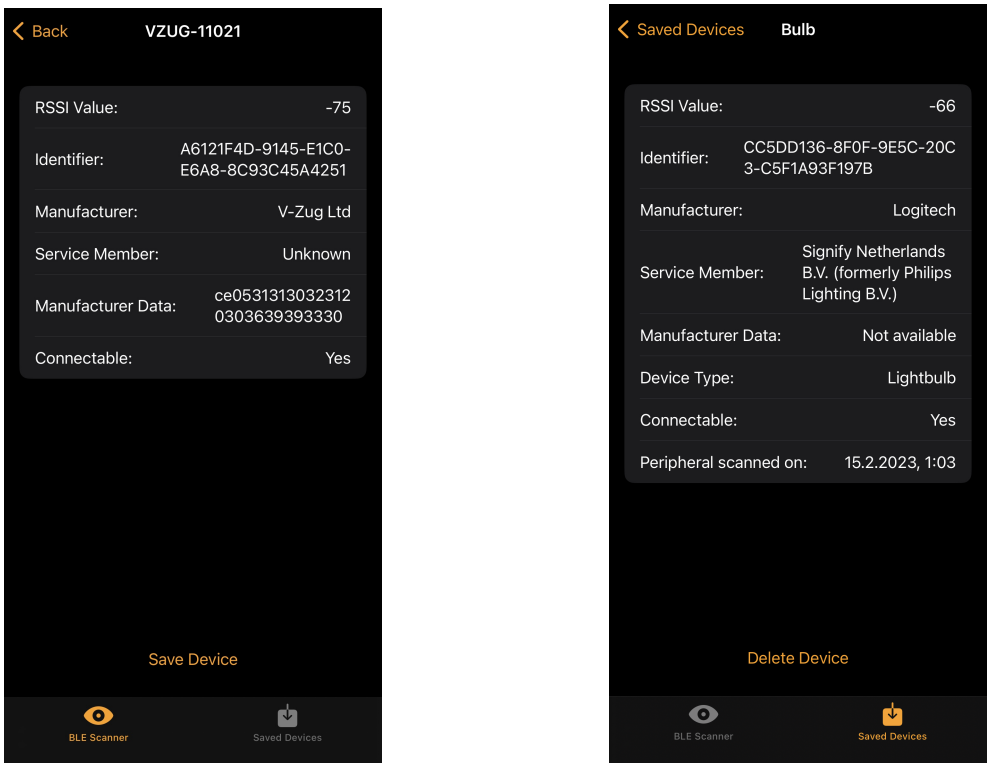
Figure 5.2: MainView

5.1.8 SavedView.swift

The view SavedView is available to the user upon using the tabbar and tapping on ”Saved Devices”. Similar to ContentView, it consists of a list of devices, only this time these are the ones that were saved inside of the database. As the application does not preserve any information, the database needs to be up and running in order to show any entries.

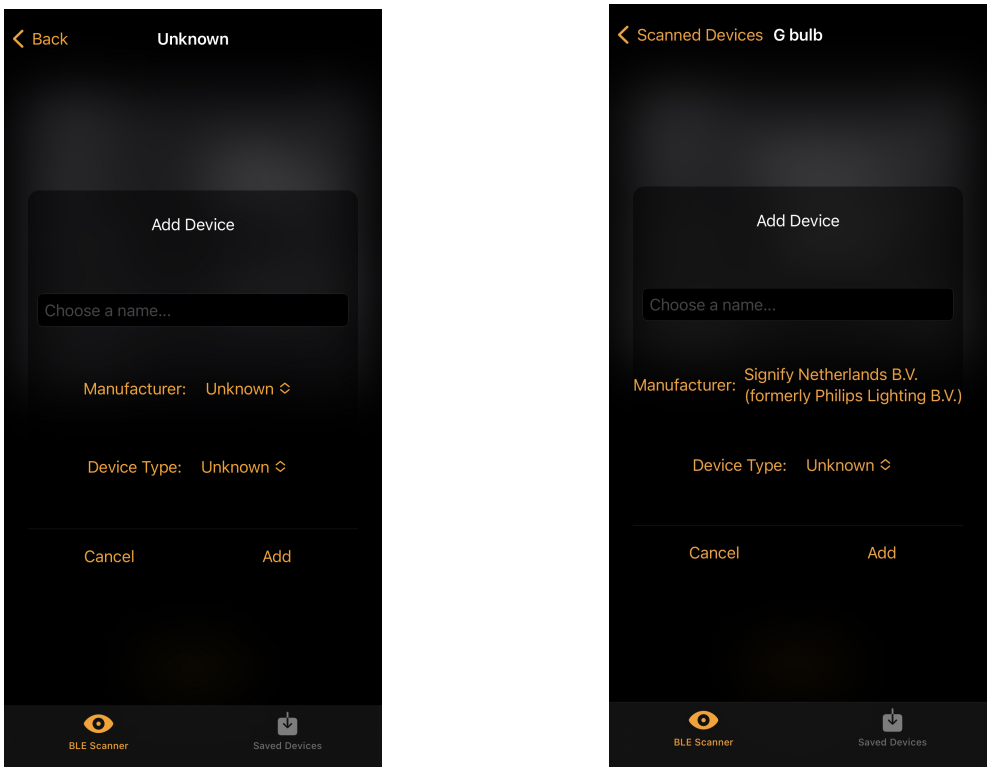
5.1.9 PeripheralView.swift

This file consists of two individual structs: The AddView and the PeripheralView. The latter is used whenever a device is inspected more thoroughly. It differentiates between a saved device and one which was only discovered but not saved yet. Figure 5.3a, picturing a device named ”VZUG-11021”, is initiated from ScannerView and features data found about the device such as the manufacturer data or if it is connectable.
The device named ”bulb” on the other hand in figure 5.3b was already saved and adds some more data. It additionally lists the device type as well as the time when the device was scanned on. On the bottom is the possibility to delete the device permanently from the database. The exact opposite can be done via the ”Save Device” button in figure 5.3a. When activated it triggers the other view that is featured inside of the PeripheralView file, which is named AddView (figure 5.4). It allows for a change of properties of the device before saving it. The name can be chosen freely, as well as the manufacturer if it was not already determined beforehand, as seen in figure 5.4. Lastly, the device type can be chosen freely from a list of options.



(a) Scanner (b) Database

Figure 5.3: PeripheralView



(a) Unknown Manufacturer (b) Recognised Manufacturer

Figure 5.4: AddView

5.2 Server/Back-end

This section will go over all files that the back-end consists of and explains the important code snippets. This GitHub repository was used as a basic template [22].

5.2.1 main.swift

This file instantiates the app and was not changed, as Vapor handles it on its own.

5.2.2 configure.swift

The function configure(_ app: Application) is called by the main.swift file when starting the application. It sets up the MongoDB client to use globally. The input is the connection URL where the database is hosted, as the Raspberry Pi is the host in this application, and the URL is the one that it uses in the current WiFi. As mentioned before it eases the usage if the IP address the Raspberry Pi uses is static, if not it might change and the connection between the server and the database fails. The iPhone, the database as well as the server share the same WiFi, which is why the internal IP address suffices.
Additionally, the function configures the hostname as well as the port. They both control incoming connections, the hostname is responsible for the address whereas the port manages which port on the before specified address accepts incoming connections. The latter can be left out as it uses the default 8080 port whereas the hostname needs to be set to the IP address where the Raspberry Pi is hosting MongoDB in order to connect it correctly.
Thirdly, the configure(_ app: Application) function is responsible for ensuring that the data which is sent and received is encoded/decoded as expected. The encoder/decoder both expect an object which conforms to the Codable protocol on one side and a JSON object on the other. At last, it calls the routes function in routes.swift.

5.2.3 routes.swift

This file handles the requests coming from the front-end. The routes function maps the get, post and delete request to the functions defined in the extension of Request. Before going over all requests, let us first have a look at the extension and what it adds.

```
self.application.mongoDB.client.db("BA")
.collection("BLEDevices", withType: BLEDevices.self)
```

It first defines a computed property which consists of all devices that are saved inside of the database. The database is called BA and the collection which is targeted is named BLEDevices. Every object which is returned should be of type BLEDevices. The collection as well as the type share the same name but differ in every other aspect. The computed

property saves them as a MongoCollection, a struct which accepts objects which are both encodable as well as decodable. As explained in section 5.4, BLEDevices conforms to the Codable alias and therefore to both the Encodable and Decodable protocol.

```
func getIDFilter() throws -> BSONDocument {
    guard let idString = self.parameters.get("_id", as: String.self) else {
        throw Abort(.badRequest, reason: "Request missing _id for device")
    }
    guard let _id = try? BSONObjectID(idString) else {
        throw Abort(.badRequest, reason: "Invalid _id string \(idString)")
    }
    return ["_id": .objectID(_id)]
}
```

The getIDFilter() function returns a BSONDocument and is used in the application when a specific entry needs to be deleted. It takes the "_id" parameter, which is a string and creates a BSONDocument with the field "_id" and the parameter as a value with the type ObjectID.

```
 func findDevices() async throws -> [BLEDevices] {
    do {
        return try await self.devicesCollection.find().toArray()
    } catch {
        throw Abort(.internalServerError,
                    reason: "Failed to load devices: \(error)")
    }
}
```

The findDevices() function returns the computed property which was mentioned above. It could fail, as computed properties calculate their values dynamically, each time they are accessed [24]. That is why a possible error needs to be caught. If the call to the database succeeds, findDevices() returns the HTTP response it gets back to the caller of the function.

```
func addDevice() async throws -> Response {
    let newDevice = try self.content.decode(BLEDevices.self)
    do {
        try await self.devicesCollection.insertOne(newDevice)
        return Response(status: .created)
    }
    catch {
        throw Abort(.internalServerError,
                    reason: "Failed to save new Device: \(error)")
    }
}
```

This function adds a BLEDevice type object to the collection by first decoding the incoming request of BLEDevices. It then calls the function to insert the device, which is defined as an extension of MongoCollection and encodes the provided values to BSON before inserting it into the database. As a unique identifier is already provided, it does not generate one. That is why the property "_id" is defined as a BSONObjectID type, as it can be used as the primary key for the document. If no errors are thrown, a created Response will be returned.

```
func deleteDevice() async throws -> Response {
    let idFilter = try self.getIDFilter()
    do {
        guard let x = try await devicesCollection.deleteOne(idFilter)
        else {
            throw Abort(.internalServerError, reason: "...")
        }
        guard x.deletedCount == 1 else {
        throw Abort(.notFound, reason: "No device with matching _id")
        }
        return Response(status: .ok)
    } catch {
        throw Abort(.internalServerError,
                    reason: "Failed to delete device: \(error)")
    }
}
```

Lastly, there is the deleteDevice() function. It first calls the getIDFilter() function and then deletes the document which has the same "_id" value using another MongoCollection function. The result is checked afterwards, as it represents the number of documents which are removed from the database and therefore should only be equal to 1. Only if that matches as well the function returns the .ok Response.

This covers all that happens in the Request extension. That this matter is shown in the routes function below as it relies heavily on the functionality of the extension. As Swift is a user-friendly language, the return statement can be omitted if a function consists of only one line as is seen here [24].

- Get Request

```
app.get { req async throws -> [BLEDevices] in
    try await req.findDevices() }
```

The app.get closure must return a list of BLEDevices and it is doing that by calling the findDevices function of the Request req. This is the function described above and returns the requested array of BLEDevices, if it is successful. The array is then returned to the front-end.

```
 app.post { req async throws -> Response in
try await req.addDevice() }
```

This closure simply calls the addDevice() function above. It returns the Response given by it to the front-end.

```
 app.delete(":_id") { req async throws -> Response in
try await req.deleteDevice() }
```

This time a dynamic parameter is used. "_id" is the parameter that will be different for every object and used as the primary key to delete the device in the database.

In the end, the BLEDevices struct gets extended by the Content protocol, to allow it to be decoded from the incoming request, which is used in the addDevice() function. It also enables the app.get closure to return the BLEDevices array. This wraps up the back-end part of the application.

5.3 Database

After installing MongoDB using the installation guidelines in the Appendix A.2, the devices can be saved in the database with the help of the back-end and Vapor. Due to privacy reasons, the database can only be accessed in a wireless local area network (WLAN) which must be set up first.


```
{
    "rssi":-94,
    "scannedOn":{"$date":"2023-01-22T21:12:23.721Z"},
    "manufacturer":"Sony",
    "serviceMember":"Unknown",
    "_id":{"$oid":"63cda6b7120e6d70b54d1635"},
    "connectable":"Yes",
    "manufacturerData":"7500420401016
    ↪ f543ad6708c05563ad6708c0401000000000000",
    "deviceType":"Television",
    "name":"My tv",
    "identifier":{"$binary":{"base64":"8VneU6AetPc10U9GSmSPXg=="
    ↪ ,"subType":"04"}},
    "isSaved":false
    }
```


That is one example of how a saved device is represented in the database. As discussed in section 4.3.2, all field values (before the :) are Strings whereas the values differ:

- rssi

  This field contains the RSSI value in the Integer format.

- manufacturer/serviceMember/connectable/manufacturerData/deviceType/name

  All these fields consist of String values.

- isSaved

  isSaved contains a Boolean value.

- _id/scannedOn/identifier

  All of these lead to an embedded document. ScannedOn and _id are created automatically when Date/ObjectID-object is added. This is the result of the initializer featured in section 5.4. The identifier field leads to the UUID of the specific peripheral.

5.4 Shared Resource: BLE Devices Package

As Mahar suggested, [25] a SwiftPM package was created to share the same model between the front-end and the back-end. The goal is to easily handle changes to the model and therefore the data representation without having to update it twice. Although this approach also has its drawbacks: It is not possible to open both the front-end as well as the back-end in XCode at the same time and expect it to compile. As both try to build their dependencies, one fails as the package BLEDevices is already occupied with the other one. That is the reason why the back-end is built and run using the command line.

```
public struct BLEDevices: Identifiable, Codable, Equatable {

    public let id: BSONObjectID

    public let identifier: UUID

    public var name: String

    public let rssi: Int

    public var manufacturer: String

    public let serviceMember: String

    public let scannedOn: Date

    public var isSaved: Bool
```

```
    public var connectable: String

    public var manufacturerData: String

    public var deviceType: String

    private enum CodingKeys: String, CodingKey {
        case id = "_id", identifier, name, rssi, manufacturer,
            ↪ serviceMember, scannedOn, isSaved, connectable,
            ↪ manufacturerData, deviceType
    }

    public init(id: BSONObjectID = BSONObjectID(), identifier:
        ↪ UUID, name: String, rssi: Int, manufacturer: String,
        ↪ serviceMember: String, isSaved: Bool, connectable:
        ↪ String, manufacturerData: String, deviceType: String)
        ↪ {
        self.id = id
        self.identifier = identifier
        self.name = name
        self.rssi = rssi
        self.manufacturer = manufacturer
        self.serviceMember = serviceMember
        self.scannedOn = Date()
        self.isSaved = isSaved
        self.connectable = connectable
        self.manufacturerData = manufacturerData
        self.deviceType = deviceType
    }
}
```

This is the struct which is used. BLEDevices conforms to three protocols: Identifiable, Codable and Equatable. The first one is used to help SwiftUI distinguish between different objects of type BLEDevices. It ensures that a unique identifier is implemented, in this case, the id property. That allows lists of BLEDevices to be used in views without having to rely on any other property, which is not necessarily unique. If a struct conforms to the Identifiable protocol, Swift automatically uses the id property for uniquing [26].
The second protocol, Codable, combines both Encodable and Decodable in a typealias, which ensures a working transformation of the provided data. In this thesis to decode from Extended JSON to a BLEDevices object and vice versa.
Lastly, BLEDevices implements the Equatable protocol as well. This is needed to compare instances of BLEDevices to each other in section 5.1.5, in order to retrieve the index where a certain peripheral is located inside of a collection. Apart from implementing those protocols, BLEDevices also includes an enum called CodingKeys. This is used to map the properties coming from a BSON format to the one of BLEDevices. It could be omitted if each property matches, but because a BLEDevices object needs to have an id property, due to confirming to the Identifiable protocol. A document in MongoDB needs

a field to be _id though, that is why the mapping is needed. Lastly, BLEDevices defines some properties, which represent the details that are saved from the scanned devices. Those are explained in table 5.1 [27] [21].

This wraps up the overview of the BLEDevices struct, which is shared between the front-end and the back-end.

Property name	Type	Definition
id	BSONObjectID	Unique identifier of each object, automatically handled by calling BSONObjectID(), easy to use with MongoDB
identifier	UUID	Unqique identifier of the saved peripheral, automatically generated by iOS
name	String	Represents the name that was given to the peripheral before saving
rssi	Int	Rssi value returned by the didDiscover function
manufacturer	String	Company of the saved device if there was a match or the user chose one
serviceMember	String	Represents the company which maps to the member service UUID and/or to the service data
scannedOn	Date	Date and time when the peripheral was detected, automatically generated upon creating an instance
isSaved	Boolean	Informs whether the device was already saved or not
connectable	String	Represents whether the peripheral allows connections, retrieved from didDiscover function
manufacturerData	String	Value of CBAdvertisementDataManufacturerDataKey
deviceType	String	Type of the peripheral chosen by the user

Table 5.1: Properties of BLEDevices

5.5 Experiments

Here are the tables listed, which resulted from the experiments. The first two are lists of devices which were inspected in a controller environment. Each was scanned four times and table 5.2 maps the outcome. Table 5.3 dissects the advertising data and lists the most important parts of it to help categorise the device. The last table (5.4) presents a summary of the essential data gathered during the public environment experiment.

Device Name	Initial Scan	1h later	5h later	1 day later
Chipolo (unconnected)	4f400707 -683B	Same	Same	F2C64BC3 -AC39
Chipolo (connected)	FD4AF4OC -1BB8	D8E7800E -AD03	D8E7800E -78BF	AFEF4A2B -B6EC
SmartTag (unconnected)	4A45C908 -B73A	Same	Same	Same
Trackpad	-	-	-	-
Tile (connected)	F159DE53 -A01E	Same	Same	Same
AirTag (connected)	AAA3CC78 -9775	Same	Same	FB7B84CC -1ABC
Apple Keyboard	-	-	-	-
FitBit	DBADC603 -2697	Same	Same	Same
Washing Machine	6121F4D -9145	Same	Same	Same

Continued on next page

Device Name	Initial Scan	1h later	5h later	1 day later
Speaker	-	-	-	-
Rii RM200	-	-	-	-
Mi Temperature Monitor	396291E9 -3F2F	Same	Same	Same
Realme 9	-	-	-	-
Apple Magic Trackpad 2022	-	-	-	-
Apple Magic Keyboard	-	-	-	-

Table 5.2: Identifier Analysis, Controlled Environment

Device Name	Manufacturer Data	Service UUIDs	Service Data
Chipolo (connected)	-	-	-
SmartTag (unconnected)	-	FD59	FD59
Tile (connected)	-	FEED	FEED
AirTag (connected)	-	-	-
FitBit	-	128-bit UUID	180A
Washing Machine	CE0531331 -C384	-	-
Mi Temperature Monitor	-	-	FE95

Table 5.3: Advertisement Data Analysis, Controlled Environment

Location	Zurich Central Station - Upper level (02:45 PM)	Zurich Central Station - Lower level (02:52 PM)	Zurich Central Station - Upper level (03:37 PM)	Company Headquarters
Scanned Devices	1000	1007	1003	176
Available Manufacturer Data	42	42	29	57
Known Localname	15	19	18	9
Available Service Data	36	49	31	22
Is Connectable	587	582	561	82

Table 5.4: Public Environment

Chapter 6

Evaluation

In order to evaluate the application, two experiments were carried out. One was conducted under controlled circumstances, where the detected advertising packets could be clearly assigned to a device. The other experiment, however, was carried out in a public place, where many more devices could be detected, but the matching was impossible. This chapter evaluates the results shown in section 5.5 in detail, first going over the general detection of IoT devices before trying to identify and categorise them. Lastly, saving the peripherals inside of the MongoDB database is tested as well.

6.1 Detection of IoT Devices

Detecting the peripherals is done by first switching Bluetooth on, before starting the scanner by interacting with the UI and starting the scanning process. As explained in section 4.1.1, whenever a peripheral was found, the didDiscover function is called immediately. In order to monitor the UUID of the peripherals in figure 5.2, the devices were viewed in isolation over a period of a day. The entries stand for the first part of their respective identifier and if it changed throughout the experiment. For seven of them, the application was not able to detect them in the first place, they must have been set to the nondiscoverable mode or are not advertising at all even though they were turned on. All others were discovered. The discovery state was consistent, there was no device that changed from nondiscoverable to discoverable or vice versa.

In the second experiment, it is not possible to say how many devices were not detected, as due to the setup, it can not be extracted how many devices were advertising but not discoverable.

6.2 Identification of IoT Devices

The next step is identifying the devices. It would be useful to know if a specific peripheral was already discovered before, for example. In the experiment with a controlled

environment, the identifier can be analysed to gather more information. Due to the implementation of the privacy options concerning the advertising address, it could be possible to detect the exact same peripheral but the identifier changed.

Some of the devices, namely the SmartTag, Tile, FitBit, Washing Machine and Mi Temperature Monitor did not change their identifier throughout the whole duration of the experiment. This means that the UUID, which is generated by iOS using the address of the peripheral, did not change (section 4.1.1) and the address must be random static at least, maybe even public. That can be used to recognise those devices over a longer period of time. Especially the Tile is surprising and quite dangerous, due to its functionality as a tracker. The same can not be said about the AirTag and the Chipolo, both when they were in a connection with another device or unconnected. The identifier did change after one day and is then impossible to recognise for the application. Their address is most likely random resolvable, as connecting with the paired device must be possible. Interesting is the duration until the address changed. In both instances, it took longer than the recommended 15 minutes.

Other types of data such as the local name or the manufacturer data can not be used to identify a peripheral, as they do not have to be unique. Many devices may advertise the same manufacturer data or service data.

6.3 Categorisation of IoT Devices

To categorize the peripherals, all data coming back from the scans was analysed. The most promising parts were the manufacturer data, the data service UUIDs and the service data. This section will investigate all of them.

6.3.1 Manufacturer Data

As discussed in section 4.1.1, the first two bytes represent the company identifier of the peripheral. Unfortunately, it proved to be rather difficult to get manufacturer data at all in the controlled experiment. If the manufacturer data starts with the company identifier of Apple (0x004C), iOS strips it away and developers do not have access to it (section 4.1.1). But also peripherals from other manufacturers did not seem to advertise any manufacturer data, except for the washing machine. Due to its manufacturer data starting with 0xCE05, its manufacturer can be retrieved [14]. It is the V-ZUG Ltd company. The identifier was then added to the BLEManufacturer enum in the application (section 5.1.6) to recognise any device that advertises that company identifier in the future. This approach was also used in the public experiment, where more devices advertised their manufacturer data, but the validity could not be verified due to its setup (figure 5.4). The BLEManufacturers enum was updated nonetheless, as the company identifications are valid. However, it can not be proven that the devices were in fact from the company its manufacturer data specifies. To counteract the uncertainty, different parts of the advertising data can be used in combination. Figure 6.1 is used as an example. The first 2 bytes of the manufacturer

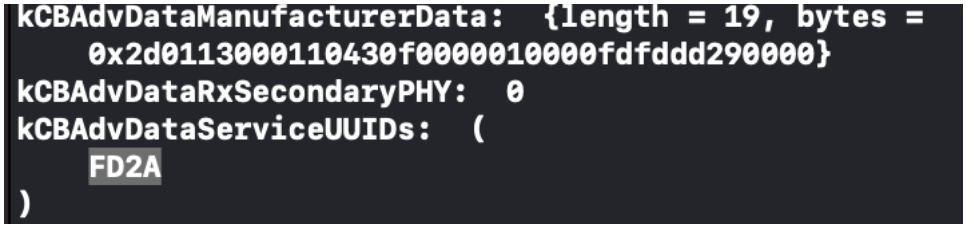


Figure 6.1: Scan result example in public environment

data point to Sony being the company behind that product. The service UUID FD2A increases the credibility of that statement, as it also points to Sony as the member company behind that service UUID.

The second experiment was used as a test in order to see how many peripherals do advertise their manufacturer data or rather how many instances are shown to the application. The results varied greatly depending on the location. In Zurich Central Station the received amount of manufacturer data was between 3-4%. In the headquarters of a company on the other hand advertised over a third of all peripherals their manufacturer data. It behaves similarly with the available localname and information about the available services. This is mostly due to the fact that the company is working in the field of IoT and therefore owns many devices for testing purposes. Interestingly enough, the peripherals were connectable in about 50% of the cases in every experiment. Nonconnectable means that a device is broadcasting. But only in a fraction of the cases data was actually showed in the UI, which leads to the assumption that some data must be stripped away by iOS. Knowing that this certainly occurs whenever the advertiser is an Apple product, it could be assumed that this is the reason behind the (in-)equality in the experiments.

Apart from the first two bytes, the vast majority of the manufacturer data could not be used for identification purposes, as each manufacturer chooses the content individually as seen in section 3.2.2. There are no official regulations on what needs to be implemented or a blueprint on how to do it.

6.3.2 Data Service UUIDs & Service Data

The idea behind the analysis of the data service UUIDs and the service data was to extract information about the functionalities of the peripheral. Due to time constraints, it is not fleshed out at the moment. It might work on certain peripherals, an example is the Tile tracker as seen in figure 6.2, where the member company of the service UUID FEED corresponds to a byte sequence starting with 0x0200. This value is declared by the Bluetooth SIG to be in the range of tags. But there are also examples where the service

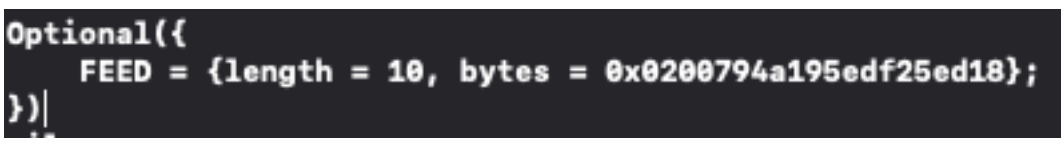


Figure 6.2: Service data of the Tile tracker

```
Optional({
    FE95 = {length = 12, bytes = 0x30585b0573ed4ac338c1a408};
})
```

Figure 6.3: Service data of the Mi Temperatur Monitor

```
Optional({
    0D00 = {length = 3, bytes = 0x4810d9};
})
```

Figure 6.4: Service data of a device, public experiment

UUID does not lead to a meaningful byte sequence, as seen with the Mi Temperature Monitor in figure 5.3. It does not provide any service UUIDs, but has FE95 as the only key of a dictionary. Xiaomi is the member company referred to by FE95, but as the value starts with 0x3058, no category can be assigned. If it would follow the same rules as the Tile, the first two bytes should be between 0x0300 and 0x033F. The device named FitBit does have a 128-bit UUID but can not be linked to any member company. The service data 180A refers to a general Device Information service in the Bluetooth SIG documentation, but this does not help to categorise the FitBit.
Therefore, the automatic detection of a device type is left out of the application and can be chosen by the user upon adding the peripheral to the database (figure 5.4). If a service UUID can be assigned to a member company, it is saved in "Service Member".

Lastly, figure 6.4 shows an example of how such a scanner can breach privacy. The service UUID "0D00" is assigned to a company named Dexcom, which produces glucose monitoring systems [28]. These kinds of devices are mostly used by diabetics to monitor their sugar levels. This kind of information is private and should not be publicly advertised. Luckily, it seems like it is not possible to retrieve much information about the devices in the vicinity, at least not with an iOS application. After scanning over 3000 peripherals, only a fraction of them provided enough information to confidently categorise them (figure 5.4).

6.3.3 Database

The connection worked without any problem for 1000 devices, it completed the whole process in about 40 seconds. It even works while the application is running in the background.

Chapter 7

Conclusions and Next Steps

In conclusion, the results of this thesis indicate that it is difficult to get much information about BLE peripherals on iOS as a developer. The vast majority of scan results during the experiments did not include the localname, the manufacturer-specific data, any service UUIDs or service data. In the aspect of security and privacy, this is positive and desirable. But when advertising data is available, the manufacturer must be cognisant of the potential risks that may be imposed upon its customers, as seen with the glucose monitoring system.

The detection of the peripherals did work, iOS allows scanning and the limiting factors seem to be on the advertiser's side, like whenever it uses the non-discoverable mode. Concerning the identification part, it is difficult to answer due to the limited devices to test the application against, but the controlled experiment showed that there are peripherals which can be recognised, as the identifier does not regularly change. As further steps, finding types of peripherals or even manufacturer who advertise their advertising data static and publicly would be interesting.

It is possible to retrieve the company of a peripheral, as long as its advertising data contains manufacturer data. The device type might be worth looking further into, as the first experiment suggests that some devices do implement the service data structure which is declared in the Bluetooth SIG documentation. The member of a data service UUID can also be used to help identify the manufacturer.

The application is able to send and receive data from an external source without any problems. Further work on the application could be improving the UI and UX by adding settings for the user, or the possibility to group devices together in the database part. Additionally, the security of the connection between the different part of the system could be improved, to allow access to the database without having to be in a LAN.

Another promising area of future research is on top of scanning the advertising data, the application could initiate a connection in order to receive more information about the device, such as all its services.

Abbreviations

BLE	Bluetooth Low Energy
CRC	Cyclic Redundancy Check
GAP	Generic Access Profile
GATT	Generic Attribute Profile
GFSK	Gaussian Frequency Shift Keying
HCI	Host Controller Interface
iOS	iPhone Operating System
IoT	Internet of Things
LAN	Local Area Network
MVVM	Model View View-Model
UI	User Interface
UUID	Universally Unique Identifier
UI	User Experience
VNC	Virtual Network Computing

List of Figures

List of Tables

Bibliography

[1] Alexander Heinrich, Milan Stute, and Matthias Hollick. "BTLEmap: Nmap for bluetooth low energy". In: *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. 2020, pp. 331–333.

[2] Alexander Heinrich, Milan Stute, and Matthias Hollick. "OpenHaystack: a framework for tracking personal bluetooth devices via Apple's massive find my network". In: *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. 2021, pp. 374–376.

[3] Jimmy Briggs and Christine Geeng. "BLE-Doubt: Smartphone-Based Detection of Malicious Bluetooth Trackers". In: *2022 IEEE Security and Privacy Workshops (SPW)*. IEEE. 2022, pp. 208–214.

[4] Alexander Heinrich, Niklas Bittner, and Matthias Hollick. "Airguard-protecting android users from stalking attacks by apple find my devices". In: *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. 2022, pp. 26–38.

[5] Johannes K Becker, David Li, and David Starobinski. "Tracking Anonymized Bluetooth Devices." In: *Proc. Priv. Enhancing Technol.* 2019.3 (2019), pp. 50–65.

[6] Cornelia Győrödi et al. "A comparative study: MongoDB vs. MySQL". In: *2015 13th International Conference on Engineering of Modern Electric Systems (EMES)*. IEEE. 2015, pp. 1–6.

[7] Zachary Parker, Scott Poe, and Susan V Vrbsky. "Comparing nosql mongodb to an sql db". In: *Proceedings of the 51st ACM Southeast Conference*. 2013, pp. 1–6.

[8] Sherali Zeadally, Farhan Siddiqui, and Zubair Baig. "25 years of Bluetooth technology". In: *Future Internet* 11.9 (2019), p. 194.

[9] R. Heydon. *Bluetooth Low Energy: The Developer's Handbook*. Pearson Always Learning. Prentice Hall, 2012. ISBN: 9780132888363. URL: https://books.google.ch/books?id=Ag2tXwAACAAJ.

[10] B. Van Der Pol. "Frequency Modulation". In: *Proceedings of the Institute of Radio Engineers* 18.7 (1930), pp. 1194–1205. DOI: 10.1109/JRPROC.1930.222124.

[11] Sabih H Gerez. "Implementation of digital signal processing: Some background on GFSK modulation". In: *Dept. Elect. Eng., Univ. Twente, Enschede, The Netherlands, Tech. Rep* (2013).

[12] Mahdi N Ali and Mohamed A Zohdy. "Interactive kalman filtering for differential and gaussian frequency shift keying modulation with application in bluetooth". In: (2012).

[13] Guillaume Celosia and Mathieu Cunche. "Saving private addresses: An analysis of privacy issues in the bluetooth-low-energy advertising mechanism". In: *Proceedings*

of the 16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services. 2019, pp. 444–453.

[14] Bluetooth Special Interest Group. *Assigned Numbers - Bluetooth Document*. 2022.

[15] Karolina Banach and Maria Skublewska-Paszkowska. "Comparison of Objective-C and Swift on the example of a mobile game". In: *Journal of Computer Sciences Institute* 16 (2020), pp. 305–308.

[16] Rob Busack. *SwiftUI and the Evolution of Apple UI Layouts*. https://www.genui.com/resources/swiftui-and-apple-layout-paradigms, visited 2023-01-04. 2020.

[17] Apple. *Core Bluetooth Documentation*. https://developer.apple.com/documentation/corebluetooth, visited 2022-10-22.

[18] Tim Condon et al. *Server-Side Swift with Vapor*. Hudson Heavy Industries, p. 232.

[19] J Dean Brock, Rebecca F Bruce, and Marietta E Cameron. "Changing the world with a Raspberry Pi". In: *Journal of Computing Sciences in Colleges* 29.2 (2013), pp. 151–153.

[20] Docker. *Docker Documentation*. https://docs.docker.com, visited 2022-10-12. 2021.

[21] MongoDB. *MongoDB Documentation*. https://www.mongodb.com/docs/, visited 2023-01-05. 2022.

[22] Martin Mahar and Carbonell Lemeden. *MongoDB + Vapor integration*. https://github.com/mongodb/mongo-swift-driver/tree/main/Examples/FullStackSwiftExample, visited 2023-01-05. 2022.

[23] Gualtier Malde. *peripheral.name vs advertisementData(kCBAdvDataLocalName)*. https://developer.apple.com/forums/thread/72343, visited 2023-01-05. 2018.

[24] Paul Hudson. *Swift for Complete Beginners*. Hudson Heavy Industries, p. 232.

[25] Kaitlin Mahar. *Building a Full Stack application with Swift*. https://www.mongodb.com/developer/code-examples/swift/full-stack-swift/, visited 2023-01-05. 2022.

[26] Paul Hudson. *Hacking with iOS: SwiftUI Edition*. Hudson Heavy Industries, p. 804.

[27] Apple. *Apple Swift Documentation*. https://developer.apple.com/documentation/swift, visited 2022-11-07.

[28] Dexcom. *Docker Website*. https://www.dexcom.com/de-DE/dexcom-de-home, visited 2022-10-12. 2023.

[29] Brian Boucheron. *Initial Server Setup with Ubuntu 20.04*. https://www.digitalocean.com/community/tutorials/initial-server-setup-with-ubuntu-20-04, visited 2023-01-05. 2021.

[30] Huobur. *How to Setup WiFi on Raspberry Pi 4 with Ubuntu 20.04 LTS 64-bit ARM Server*. https://huobur.medium.com/how-to-setup-wifi-on-raspberry-pi-4-with-ubuntu-20-04-lts-64-bit-arm-server-ceb02303e49b, visited 2023-01-05. 2020.

[31] Mark Drake and finid. *How to Install and Configure VNC on Ubuntu 20.04*. https://www.digitalocean.com/community/tutorials/how-to-install-and-configure-vnc-on-ubuntu-20-04, visited 2023-01-05. 2021.

[32] Mark Smith. *Install & Configure MongoDB on the Raspberry Pi*. https://www.mongodb.com/developer/products/mongodb/mongodb-on-raspberry-pi/, visited 2023-01-05. 2022.

[33] Shanika Wickramasinghe. *How To Run MongoDB as a Docker Container*. https://www.bmc.com/blogs/mongodb-docker-container/, visited 2022-10-12. 2020.

Appendix A

Installation Guidelines

These guidelines serve as a template to get the application running on other devices. It is assumed that the Raspberry Pi is brand new and that Xcode was already installed.

A.1 Raspberry Pi - Initial Setup

In order to set the Raspberry Pi up, a microSD USB reader is needed except the operating system used on the Raspberry Pi is Ubuntu 20.04. If that is the case and the internet connection is already set up, steps 1-7 can be skipped. Installing a Virtual Network Computing (VNC) and ensuring it starts when the server boots up, which starts at step 11, is not necessary but it facilitates the process as everything after can be done on one computer instead of jumping between the Raspberry Pi and the client device(s) [29][30][31].

1. Flash the microSD card with the help of a computer and the official Raspberry Pi Imager. Make sure to install the correct operating system: Ubuntu 20.04.

2. Insert the card into the Raspberry Pi and boot it up.

3. Log in using "ubuntu" as the username as well as the password. After that, it is possible to change the password, which is highly encouraged due to privacy reasons.

4. Make sure to have the WiFi card name ready, which can be found by typing

   ```
   $ ls /sys/class/net
   ```

   into the terminal and looking for a wlan prefix.

5. Open the 50-cloud-init file as an administrator, using the sudo nano command. The file is stored in /etc/netplan. You may need to input the password as you need admin access.

```
# This file is generated from information provided by the datasource.  Changes
# to it will not persist across an instance reboot.  To disable cloud-init's
# network configuration capabilities, write a file
# /etc/cloud/cloud.cfg.d/99-disable-network-config.cfg with the following:
# network: {config: disabled}
network:
    ethernets:
        eth0:
            dhcp4: true
            optional: true
    version: 2
```

Figure A.1: 50-cloud-init.yaml

6. Edit the file like in figure A.1. ”wlan0” is your WiFi card name from step 4, ”MyWiFi” is the name of your chosen router and ”MyPass” it’s associated password. Please make sure that the layout is exactly the same as in figure A.1 and that the indentations consist of spaces as tabs do not work.

7. Save the file. After rebooting the internet connection should be set up, test it via the terminal:

   ```
   $ ip a
   ```

   An IP address in the section of the WiFi card name should be visible.

8. Next a user needs to be created. First change to the root user before setting a password:

   ```
   $ sudo -i
   $ passwd
   ```

   Now create the user, where the username can be chosen freely.

   ```
   $ adduser username
   ```

   From now on, this guide will use aljremote, as actually used in the project.

9. Allow the newly created user to access root privileges with

   ```
   $ usermod -aG sudo aljremote
   ```

   Now aljremote can use sudo before a command as well.

10. As the database will be hosted on this device with possibly highly sensitive data it should have a firewall. This is done with:

    ```
    $ ufw allow OpenSSH
    $ ufw enable
    ```

    The firewall now blocks all connections except for SSH.

11. Now a graphical desktop environment is installed as well as a VNC server to connect the Raspberry Pi to another device. First update the list of packages before installing the xfce4-goodies package:

```
$ sudo apt update
$ sudo apt install xfce4 xfce4-goodies
```

The chosen default display manager is not relevant.

12. After the installation process has finished, install the TightVNC server:

```
$ sudo apt install tightvncserver
```

13. Set up the server with

```
$ vncserver
```

and input a secure password to access the Raspberry Pi remotely. Be aware of the fact that the password length must be between 6 to 8 characters. A view-only password can also be chosen if desired but it is not necessary. Afterwards, TightVNC launches the default server instance on port 5901.

14. On startup, the VNC server needs to know to which graphical desktop environment it should connect to. First kill the instance that is running

```
$ vncserver -kill :1
```

The xstartup file you need to change is located in the .vnc folder and it should look like this:

```
#!/bin/bash
xrdb $HOME/.Xresources
startxfce4 &
```

15. Restart the server with

```
$ vncserver -localhost
```

16. Now any other computer on the same network can connect to the server. In order to do that, write the following command into the terminal of the computer

```
$ ssh -L 5901:127.0.0.1:5901 -N -f -l aljremote
  ↪ server_ip
```

Do not forget to change aljremote as well as server_ip to the corresponding values in your setup.

17. The server should immediately start whenever the Raspberry Pi boots up, that is why a unit file is created with

```
$ sudo nano /etc/systemd/system/vncserver@.service
```

Edit the file that it look similar to this

```
[Unit]
Description=Start TightVNC server at startup
After=syslog.target network.target
```

```
[Service]
Type=forking
User=aljremote
Group=aljremote
WorkingDirectory=/home/aljremote

PIDFile=/home/aljremote/.vnc/%H:%i.pid
ExecStartPre=-/usr/bin/vncserver -kill :%i > /dev/null
    ↪ 2>&1
ExecStart=/usr/bin/vncserver -depth 24 -geometry 1280
    ↪ x800 -localhost :%i
ExecStop=/usr/bin/vncserver -kill :%i

[Install]
WantedBy=multi-user.target
```

18. The file needs to be activated but not before the system notices it first

```
$ sudo systemctl daemon-reload
$ sudo systemctl enable vncserver@1.service
```

That is it with the initial setup of the Raspberry Pi how it was used in the thesis. Ubuntu 20.04 was installed together with a VNC to allow working on the server remotely. The connection should work by only booting the Raspberry Pi up and executing the command from step 17.

A.2 MongoDB

The installation of MongoDB was done with the official guide on MongoDB's website [32].

1. To start off, the MongoDB 4.4 GPG key is installed

```
$ wget -qO - https://www.mongodb.org/static/pgp/
    ↪ server-4.4.asc | sudo apt-key add -
```

2. The next step is to add the source location for the needed packages

```
$ echo "deb [ arch=amd64,arm64 ] https://repo.
    ↪ mongodb.org/apt/ubuntu focal/mongodb-org/4.4
    ↪ multiverse" | sudo tee /etc/apt/sources.list.d
    ↪ /mongodb-org-4.4.list
```

3. The package details can now be downloaded

```
sudo apt-get update
```

4. Finally the installation is one command away

```
$ sudo apt-get install -y mongodb-org
```

5. As in the Raspberry Pi setup, ensure that the correct file is picked up and MongoDB is started when the system boots up

```
$ sudo systemctl daemon-reload
$ sudo systemctl enable mongod
```

6. The main goal of this installation process is to access it afterwards from the back-end. That is the reason why MongoDB needs to be accessible in the scope of the network. In order to ensure that, the /etc/mongod.conf file needs to be edited, specifically the bindIp value should be set to "0.0.0.0".

7. After restarting mongod, open up the port that MongoDB is using

```
$ sudo systemctl restart mongod
$ sudo ufw allow 27017/tcp
```

MongoDB is now up and running as well as available on the entire network and therefore ready to work with.

A.3 Docker

Docker is installed in order to start an instance of MongoDB as soon as the Raspberry Pi is booted [33].

1. The first step is installing these prerequisites

```
$ sudo apt install apt-transport-https ca-
  ↪ certificates curl software-properties-common
```

2. before adding the GPG key for the Docker repository using this command

```
$ curl -fsSL https://download.docker.com/linux/
  ↪ ubuntu/gpg | sudo apt-key add -
```

3. Next the Docker repository is added to the APT sources

```
$ sudo add-apt-repository \ "deb [arch=amd64] https
  ↪ ://download.docker.com/linux/ubuntu $(
  ↪ lsb_release -cs) stable"
```

4. The package list needs to be updated

```
$ sudo apt update
```

5. Then the docker repository is verified before installing the community edition of Docker

```
$ apt-cache policy docker-ce
$ sudo apt install docker-ce
```

6. Docker Compose is installed next, followed by transforming it into an executable

```
$ sudo curl -L "https://github.com/docker/compose/
  ↪ releases/download/1.27.4/docker-compose-$(
  ↪ uname␣-s)-$(uname␣-m)" -o /usr/local/bin/
  ↪ docker-compose
$ sudo chmod +x /usr/local/bin/docker-compose
```

7. A directory is created called mongodb which should hold a database directory

```
$ mkdir -pv mongodb/database
```

8. Next, the container can be started with

```
$ sudo docker-compose up -d
```

9. Finally the container should immediately start whenever the system boots up

```
$ sudo systemctl enable mongod.service
```

The MongoDB database now starts upon booting and runs as a background process.

A.4 Running the Application

After the whole setup is done only a few steps are necessary to start using the application.

1. Plug the Raspberry Pi in and wait a few seconds. Optionally connect it to another screen to see when the booting process finishes.

2. The back-end needs to know the IP address where the database is hosted. That can be done in the configure.swift file by assigning the address from the database instance on the Raspberry Pi (section A.1 step 7) to the app.mongoDB.configure variable. It should look similar to this

```
try app.mongoDB.configure("mongodb
  ↪ ://192.168.0.30:27017")
```

3. Run the following command in the terminal, ensure to change aljremote and server_ip to the corresponding values in your setup.

```
ssh -L 5901:127.0.0.1:5901 -N -f -l aljremote
  ↪ server_ip
```

4. Next the back-end needs to be started. Go to the directory in which the back-end resides and execute

```
swift run
```

   It is crucial to do that via the terminal, as it does not work to run both the front-end and the back-end simultaneously using XCode...

5. Start the application on the iPhone, either through XCode or by clicking on the icon if it is already installed. Make sure the iPhone is connected to the same WiFi as the back-end.

This concludes the installation guidelines for the system used in this thesis.