University of Zurich[UZH]

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

HomeScout: A Modular Bluetooth Low Energy Sensing Android App

Louis Bienz
Zurich, Switzerland
Student ID: 15-729-403

Supervisor: Katharina Müller
Date of Submission: January 5, 2023

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

ifi

Master 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

Bluetooth Low Energy (BLE) trackers are affordable devices that are misused for stalking attacks. Apple, for example, offers AirTags as BLE trackers. These are used by users of their ecosystem to retrieve lost items. Unfortunately, cases have come to light in which owners of an AirTag have tracked other people with the help of such a tracker. Apple reacted to this to protect their users. Android users, on the other hand, were not offered a satisfactory solution. An Android app called AirGuard from the Technical University of Darmstadt in Germany addressed this issue. The app scans in the background and informs users about trackers that are following them. Apple and AirGuard use a tracking algorithm defined by them that decides whether a tracker is following the user. In this work, an Android app is developed that allows users to customize such a tracking algorithm. This also enabled this work to evaluate through experiments how such a tracking algorithm should be configured to notify users as quickly as possible of malicious trackers. Furthermore, this app applies the tracking algorithm not only to BLE trackers but to BLE devices in general, since in principle not only BLE trackers can be used for stalking attacks. Moreover, this work investigated whether BLE trackers can be distinguished from BLE devices using Android's BLE API.

Acknowledgments

First and foremost, I would like to thank Katharina Müller for her full support. As my supervisor, she was actively available to me with valuable advice, constructive feedback, and most importantly, her time. Furthermore, it was a pleasure to do my master thesis in the Communication Systems Research Group (CSG) at the Department of Informatics of the University of Zurich. The given topic was very motivating. I would also like to thank my dear partner Corina Syfrig for her patience during the last six months. She has motivated me and helped me to keep the big picture in mind. Finally, I would like to thank my friend Marc Zwimpfer and my father Charles Bienz, who critically reviewed my work and made valuable suggestions for improvement.

Contents

Chapter 1

Introduction

The intended purpose of Bluetooth Low Energy (BLE) is to enable communication between devices. Nowadays, it is also used for positioning of devices, especially indoors where the presence, distance and direction to other devices can be determined [1]. Furthermore, with Apple's Find My app, it is possible to locate BLE devices not only indoors, but globally [2]. An owner can locate an item attached to an AirTag anywhere in the world. For example, an AirTag can be attached to a travel bag. If the airline ships the bag to the wrong destination the owner is able to see that the bag is in a different country. Likewise, an AirTag can be attached to keys. If the owner loses those keys on the way home, the owner will know where in the city those keys are. Hence, an item can be located and found anywhere. Unfortunately, such AirTags have been misused for stalking purposes. The news reports cases where people have been stalked with the help of AirTags [3], [4], as well as cases where AirTags have been hidden in a car to track the person [5] or steal cars [6].

Apple has reacted to this concern and updated iOS with tracking protection. It notifies a user if an AirTag or another Find My network accessory is separated from the owner and is moving with the user, respectively the victim [7]. In case a victim does not own an iPhone, the AirTag will play a sound once it is separated from its owner. This way victims are able to detect that an AirTag is following them. However, AirPods for example are not capable of playing a sound to inform a victim. Additional countermeasures are needed to prevent stalking attacks with BLE devices in general. For Android user's the Tracker Detect app allows one to manually find such devices. The app covers all devices which are compatible with the Find My app. Furthermore, it shows how to disable such trackers [8]. To further improve the stalking or tracking protection of Android users, AirGuard has been developed. The benefit of this app is, that it scans for Find My trackers and Tile trackers in the background [9]. If the app detects a tracker which is following the victim, it sends a notification to the victim, respectively to the user of the app.

Besides Apple's Find My trackers, which include trackers from Chiplo, and Tile trackers other companies have released their own trackers as well. Samsung sells the Galaxy SmartTag+ which works in the same manner as the AirTag, but with the Galaxy Find network. Trackers from Tile are located via the Tile app. Moreover, the Tile app allows one to pair third-party accessories, such as HP Laptops, Fitness-Trackers from FitBit,

any kind of headphones from Skullcandy, and many more. Hence, not only the mentioned trackers but BLE-capable devices, in general, can be misused as stalkingware as it has been the case with AirTags.

To summarize, BLE trackers from Apple, Chipolo and Tile can be detected using the AirGuard app. At this point, no app is known that protects users from stalking attacks using a Galaxy SmartTag+. Furthermore, it is likely that more companies release BLE trackers and Tile provides the possibility to use other BLE devices as trackers. Thus, not only the mentioned BLE trackers but BLE devices, in general, can be used for stalking attacks. Therefore, this work creates an Android app that protects users from stalking attacks using BLE-capable devices.

1.1 Motivation

This work aims to extend tracking protection to the GalaxySmartTag+ as well as for BLE devices in general. Using the Android Framework to build a tracking detection app, the motivation is to see if there is a holistic way to identify BLE devices as BLE trackers and eventually classify them as malicious or non-malicious trackers. This helps stalking victims to be better protected from stalking attacks.

Furthermore, Apple's stalking protection and the AirGuard app use tracking algorithms, which classify trackers as malicious or non-malicious. The main questions for this decision are:

- For how long has a tracker been following a user?
- For how many meters has the tracker been following a user?
- How many times has a signal from this tracker occurred?

Their algorithm is enclosed and cannot be edited by a user. For this reason, the app developed for this work aims to allow the user to set those parameters individually. On one hand, this allows a user to raise or lower the tracking protection for their individual level. On the other hand, it creates the possibility to study which values fit best for a tracking algorithm with the motivation to notify users as early as possible about malicious trackers.

1.2 Description of Work

The goal of this thesis is to generate new insights into the area of BLE devices used as trackers. This work investigates if BLE-capable devices can be identified as trackers and how a tracking algorithm can be designed in order to do so. Thus, the main contributions of this work are:

(i) Investigation on holistic Bluetooth Low Energy tracker identification.

(ii) Implementation of an app to detect Bluetooth Low Energy trackers with an adaptable tracking algorithm.

(iii) Analysis of the parameters used for the tracking algorithm to notify users as early as possible.

Furthermore, additional secondary contributions are addressed within this work. The Media Access Control (MAC) address of a BLE device is regarded as a unique identifier. It helps the tracking algorithm to keep track of a device. This however might impose some issues, as nowadays some devices implement MAC randomization. Hence, it might not be possible to track the devices appropriately. Therefore, for the AirTag, Chipolo ONE Spot, Galaxy SmartTag+ and the Tile tracker additional experiments are conducted to gain insights regarding their MAC randomization patterns. This is presented in Section 5.1.1.

Finally, out of curiosity, the tracker's Received Signal Strength Indicator (RSSI) values are under investigation. The corresponding experiments evaluate how the RSSI values behave in different scenarios. The insights are described in Section 5.1.2.

1.3 Thesis Outline

This thesis is structured as follows: Chapter 2 presents related work in the area of BLE, Mac randomization, offline finding networks and existing tracking protection. Chapter 3 introduces the basics about BLE, offline finding networks, existing tracking algorithms and the trackers used in this work. Chapter 4 describes the app developed for this work, what the Android BLE API provides, and how the trackers used in this work can be identified. Chapter 5 presents the results and evaluation of the conducted experiments about MAC randomization, the tracker's RSSI values, and the tracking algorithm. Chapter 6 discusses the findings of this work. Finally, Chapter 7 closes this work with the conclusion and points out future work.

Chapter 2

Related Work

BLE is used for various applications such as wildlife animal monitoring [10], smartphone-based indoor localization [11], [12], [13], indoor localization based on the Received Signal Strength Indicator (RSSI) [14], [15], [16], [17] indoor and outdoor localization in combination with GPS [18], and offline finding networks for BLE trackers [19]. Such offline finding networks build the basis to locate BLE trackers globally. This means the owner of the tracker does not have to be within the connection range of the tracker to retrieve its location. This enables users to find their lost items even if they are far away. Unfortunately, those offline finding networks have led to stalking attacks, where a victim has been located with the use of a tracker.

With those offline finding networks, tracking a target is easy. Even in crowded places, tracking a BLE device is possible as shown by Nikodem and Bawiec [20]. In their large-scale experiment, they tracked BLE devices in a connectionless manner, i.e. using only BLE advertisements. In a laboratory of 21 m^2, they tested the reliability of BLE advertisements using 210 tags. The advertisement interval was set to 250 ms and 750 ms and the data changed every 10 s. Thus, a passive scanner should receive 40, respectively 13 advertisements with the same data. For both advertisement intervals, the data reception rate was above 99%. Nikodem and Bawiec showed that advertisement-based BLE communication scales well and has acceptable collision rates such that IoT applications are reliable. This means in a broader sense, BLE works in a large network of tags, respectively in crowded places. Therefore, it is likely that also tracking an item or a person in crowded places using an offline finding network is possible in a reliable manner.

BLE devices emit advertisements to show their presence to other devices. Those packets include the public Media Access Control (MAC) address of the device. Broadcasting its own public MAC address enables tracking of the device itself. This may pose a threat because the person mapped to this very device can be tracked. To prevent this, MAC randomization can be applied. This is a technique, where a BLE device randomizes its MAC address in its advertisements. Thus, it does not send its permanent MAC address anymore. Becker et al. [21] analyzed state-of-the-art devices implementing MAC randomization. In their experiment, they mimicked a local adversary, who is a passive external adversary only reading BLE advertisements. The adversary does not add, remove or modify any of the traffic nor does the adversary interact with the tracked device either by a

scan or connect request. In the experiment, the adversary stayed within the connection range of the device and the tracked device had Bluetooth enabled. Becker et al. were able to extract so-called identifying tokens from the advertisement payloads. They implemented an address-carryover algorithm and showed that they could track devices beyond MAC randomization based on those identifying tokens. Windows 10 systems were tracked unbounded and macOS and iOS devices were tracked for 53 min. Only Android devices showed no vulnerability.

As mentioned before, offline finding networks build the basis for locating BLE trackers. The Find My network is Apple's crowd-sourced tracking system which uses Bluetooth to locate lost devices like MacBooks, iPhones, AirTags or other Apple products. It has been introduced in 2019. In general, such an offline finding network helps to find lost devices. So-called finder devices can detect the presence of lost devices via BLE. The finder device afterwards sends its location via the internet to the owner of the lost device. Thus, the owner is able to get the location information on his or her lost devices. Besides Apple devices also non-Apple devices can be located with the Find My network, for example, Bluetooth-capable devices modified with OpenHayStack [22], which is shortly introduced in the next paragraph. Apple says that its Find My network is not accessible to Apple itself. The location data sent from a finder device to Apple's server is encrypted and deleted after 24 hours [23]. Heinrich et al. [19] analysed the security and privacy of Apple's Find My network, respectively offline finding network using reverse engineering. Based on their analysis they assessed the security and privacy of the system. They found that Apple achieves its security and privacy goals, however, they detected two vulnerabilities to the system. A correlation can be made between different locations of an owner if those locations have been reported by the same finder device. This would allow Apple to create a social graph. Besides that, Heinrich et al. found that malicious macOS applications are able to retrieve the location reports for the last seven days of the user and all of their devices without their consent.

OpenHaystack [22] is a framework to track personal Bluetooth devices using Apple's Find My network. It has been created by Heinrich et al. from the TU Darmstadt in Germany. With this framework, a user can create tracking accessories out of Bluetooth-capable devices. It provides firmware images for Nordic nRF5 chips and the ESP32. Along comes a macOS application to locate the Bluetooth devices. Both are available on GitHub[1]. The application allows the creation of new accessories to be tracked and deploys the firmware on a supported device via USB. After 30 minutes the first location reports are available via the application. The location of such a device happens via Apple's Find My network, thus the location of the accessories are uploaded to Apple's server by iPhone participating in the network.

To fight stalking attacks with AirTags, which is a BLE device, Apple updated their iOS such that users get notified about malicious AirTags following them. Furthermore, for Android users, Apple released an app in the Google Play Store called Tracker Detect in February 2022. This app allows one to manually scan for lost devices of Apple's Find My network, such as AirTags [8]. For example, Tracker Detect detects an AirTag, which is paired with an Apple ID, as soon as it loses connection to its paired account. However,

[1] `https://github.com/seemoo-lab/openhaystack`

this app does not scan for tracking devices in the background. Hence, in case the user suspects being tracked he or she needs to manually start a scan in order to identify a potential tracking device.

AirGuard is an Android App which looks for malicious trackers in the background and notifies a user if he or she is being tracked by an AirTag for example. This app has been designed and built within the work of Heinrich et al. [24]. They reverse-engineered Apple's tracking protection to identify AirTags based on their manufacturer-specific data. Furthermore, they compared their App AirGuard against Apple's built-in tracking protection. In their experiment, they tested both tracking protections against an AirTag, Chipolo ONE Spot and a self-made OpenHaystack tag. They recorded how much time has passed until the user gets notified about a malicious tracker. They evaluated three scenarios. In the first scenario, the tracking devices had been hidden in the pocket of the jacket of the target person. iOS notified the user about the AirTag and the Chipolo ONE Spot trackers after 1 h and 45 min. Regarding the self-made OpenHaystack, no notification had been sent to the user. AirGuard on the other hand notified the user about all three trackers within 35 min. In the second scenario, the trackers had been hidden in the backpack of the target person. iOS again did not notify the user about the OpenHaystack tag and sent a notification after 4 h 14 min for the AirTag and the Chipolo ONE Spot. AirGuard sent a notification after 30 min for all three trackers. In the last scenario, the trackers had been hidden in the car, more precisely on the inside of the fuel cap. After 4 h 18 min, AirGuard notified the user about all three trackers, whereas iOS left the target person uninformed, respectively did not send any notification.

To summarize, BLE is amongst other applications used to track items via an offline finding network. This is very reliable, even in crowded places. Those trackers have been misused for stalking purposes. Therefore, Apple reacted with countermeasures. The app AirGuard enhanced tracking protection to even more trackers. Both tracking protections use an algorithm which classifies trackers as malicious or non-malicious devices. Those tracking algorithms are determined by Apple and AirGuard. The user does not have the possibility to change how this tracking algorithm evaluates devices as malicious or not. Today, no app is known which lets users define the values for the tracking algorithm. Furthermore, no analysis of such a tracker algorithm has been found. Finally, the existing apps for tracking protection target only actual BLE trackers and not BLE devices in general. Therefore, for this work, an app is created to inform users about any BLE devices which might be misused as a tracker to follow and locate them. Furthermore, this work focuses on how to configure a tracking algorithm such that a user gets notified about a BLE tracker as early as possible.

Chapter 3

Background

This chapter presents the BLE Technology as it builds the foundation for locating BLE trackers using offline finding networks. How an offline finding network accomplishes locating trackers is explained by the example of Apple's Find My network. Afterwards, tracking protection and tracking devices which have been studied or used within this work are described.

3.1 Bluetooth Low Energy

The technical explanations of BLE in this section are based on the BLE Primer from the official Bluetooth Special Interest Group (SIG) [25]. Additional resources are cited accordingly.

BLE enables electronic devices to communicate in three different types of topologies. A one-to-one communication between two devices is done in a point-to-point connection-oriented or connectionless manner. This might be used to exchange data for example. A One-to-many communication broadcasts data to an unlimited number of receivers using so-called advertisements. The advertisement types are non-connectable, non-scannable, and undirected. The broadcasting method can also be extended to a mesh topology, where multiple devices in a network can communicate with any other device in the network. The mesh as well as the point-to-point topology is neglected in this work since the focus lies on the passive scanning of trackers. Thus, only the broadcasting topology will be explained in more detail.

Generally, BLE offers a raw data rate of 2 Mbit/s. It allows to exchange of data wirelessly between two devices and no additional, respectively intermediate instance is needed. BLE was designed to be highly energy efficient. This made BLE beneficial for small, wireless devices such as mice, sensors, wearables, or generally speaking, devices that run on small batteries for a longer period of time.

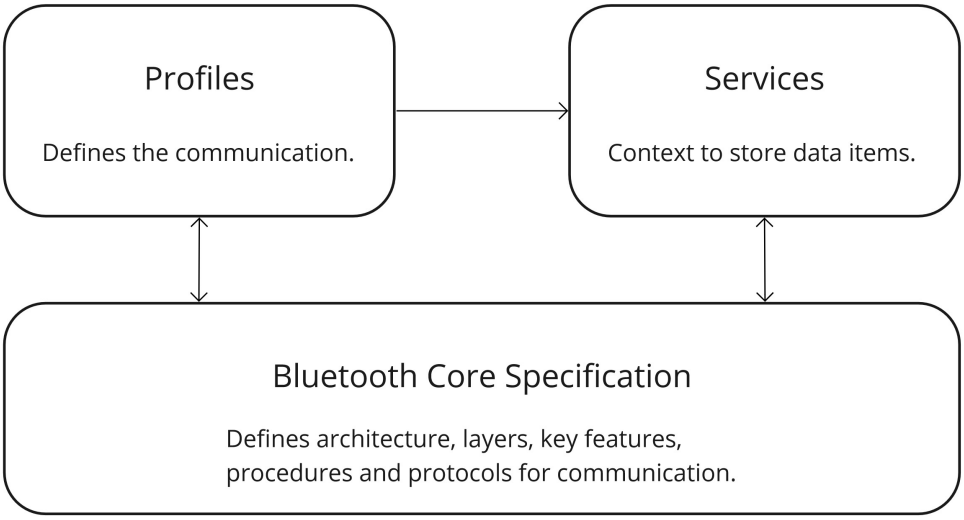

Figure 1: Schema of BLE Specifications. Adapted from [25, Figure 1].

3.1.1 Specifications

The specifications of BLE comprise three parts. The Bluetooth Core Specification defines the architecture, procedures, and protocols. For devices to be interoperable using BLE, the specifications of Profiles and Services are relevant. Figure 1 shows an overview of those three elements.

BLE Core Specification

The Core Specification is the primary specification for BLE and Bluetooth Classic. It includes the architecture, the layers, key features, procedures and protocols such that devices can communicate at the corresponding layer.

Profile Specifications

Bluetooth follows the idea of a client-server architecture when two devices are communicating. A server stores data and a client retrieves data from it. From this point of view and for this work, a tracking device is regarded as a server and a smartphone trying to detect a tracking device is regarded as a client. The Profile Specifications define how such a client-server communication works. Depending on the application an appropriate Profile needs to be considered. A list of all profiles can be found on the official Website of the Bluetooth SIG [2].

[2] `https://www.bluetooth.com/specifications/specs/`

Service Specification

A Service provides context for so-called Characteristics and Descriptors. They formally define data items which are stored on a server. All services can also be found on the official Website of the Bluetooth SIG [2].

Assigned Numbers

Services, Characteristics and Descriptors have a universally unique identifier (UUID). It identifies the type of service, characteristic or descriptor. As an example, some profiles require a company to identify itself. A list of all the Company Identifiers is provided by the Bluetooth SIG [26].

3.1.2 Layer Stack

The OSI reference model conceptualizes how to communicate between heterogeneous systems. The BLE Layer Stack spans over all those levels from the OSI reference model. This makes BLE independent from external bodies. The BLE Layer Stack is further divided into a Host and a Controller which in turn are connected via the Host Controller Interface (HCI).

Physical Layer

BLE operates on the 2.4 GHz ISM Band (2.402 - 2.480 GHz). Those are the same Frequency Bands used for the Bluetooth Classic. It is split into 40 channels with 2 MHz spacing. To transmit and decode data, respectively radio signals, BLE uses Gaussian Frequency Shift Keying (GFSK) for its modulation scheme. There are further three modulation schemes defined which are called PHY. The type LE 1M has a symbol rate of 1Ms/s and must be supported by all devices. Type LE 2M, which has a higher symbol rate of 2 Ms/s and a different deviation from the channel center compared to LE 1M, is optional for a device. Lastly, type LE Coded has the same symbol rate as LE 1M but enables Forward Error Correction (FEC). It is also optional. Regarding the communication scheme, BLE represents a half-duplex mode using Time Division Duplex (TDD), making it appear as a full-duplex scheme. Furthermore, the specification defines the Transmission Power and the Receiver Sensitivity. The output power level at the maximum power setting needs to be in between 0.01 mW (-20 dBm) and 100 mW (+20dBm). The Receiver Sensitivity is defined by the Bit Error Rate (BER). The BER varies in length because in the Link Layer a Cyclic Redundancy Check (CRC) gets appended. Typical Receiver Sensitivity is a BER of 0.1%. Lastly, BLE uses two methods to calculate the angle from which a signal was sent. The Angle of Arrival (AoA) and the Angle of Departure (AoD).

Figure 2: Packet format for LE 1M and LE 2M. Adapted from [25, Figure 7].

Link Layer

The Link Layer has the largest specification. It touches Packets, States, Channels and Addresses which are of interest to this work. Thus, those are described in more detail.

Packets. The packets are transmitted over the air. Depending on the PHYs the Link Layer shows two packet types. One is used for LE 1M and LE 2M whereas the other is in place for LE Coded. The packet for LE 1M and 2M are shown in Figure 2. A figure for the coded packet is not included, since all trackers used in this work do not use LE Coded, as seen in Appendix A. A receiver uses the preamble to synchronize on the frequency of the signal. The Access Address helps to distinguish between background noise, relevant signals, or advertisements which are broadcasted to any device. The CRC helps to determine if there are errors in one or more bits. Last but not least, the Protocol Data Unit (PDU) is the part of the packet which is used for data transmission.

States. The Link Layer uses a state machine with seven different states. The states are depicted in Figure 3. An explanation of the states can be found in Table 1. A device listening for advertisements of other devices is in the Initiate state. The counterpart to this is a device in the Advertising state which broadcasts advertisements. Once the two devices have established a connection two different roles are assigned to the devices involved. The one transitioning from the Initiating State to the Connection state is regarded as the Client. The other device, which transitioned from the Advertising state to the Connection state takes up the role of the Server. In this work, a tracking device refers to a device which is in the Advertising state. A smartphone which scans for tracking devices represents a device in the Initiating state. However, no connection will be established.

Table 1: Description of the states in the Link Layer. Source: [25, p. 21]

State	Description
Standby	Device neither transmits or receives packets.
Initiating	Responds to advertisements from a particular device to request a connection.
Advertising	Transmits advertisements and potentially processes packets sent in response to advertisements by other devices.
Connection	In a connection with another device.
Scanning	Listening for advertisements from other devices.
Isochronous Broadcast	Broadcasts isochronous data packets.
Synchronization	Listens for periodic advertising belonging to a specific advertising train transmitted by a particular device.

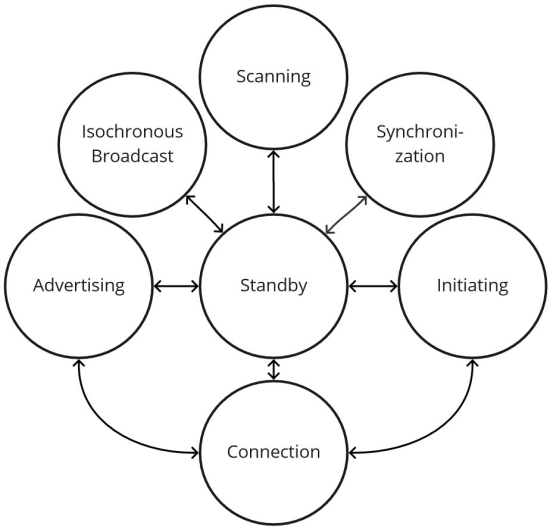

Figure 3: State Machine for States of the Link Layer. Source: [25, p. 21, Figure 9].

Channels. The Link Layer defines 40 Channels. The division of the 2.4 GHz frequency band is controlled by the link layer. Various spread spectrum techniques are used to reduce the chances of collisions. This makes communication more reliable. An example spread spectrum technique used is adaptive frequency hopping. A channel is chosen with a channel selection algorithm and a table of data called a channel map. This map states whether a channel is used or unused depending on the quality of communication. This ensures that the channel selection algorithm does not choose channels with poor quality.

Addresses. Device addresses can be used as device identifiers. It is 48-bits long and various types exist. A summary of the address types is given in Table 2.

Transport Architecture

To communicate data different schemes can be used, depending on the application. This always involves four different channels. The Physical Channel, The Physical Link, the Logical Transport and the Logical Link. The Physical Channel determines the way of communication. This can be either connected point-to-point communication or a broadcast which is connectionless. Those two different schemes differ from each other. A point-to-point connection for example involves frequency hopping across the 37 communication channels. For broadcasting, the LE Advertising Physical Channel is used. The Physical Link describes a specific physical channel and its characteristics. The Logical Link and Transport define a set of parameters which suit a particular way of data communication over the physical link using a physical channel.

As an example of the Transport Architecture, a broadcast is based on the LE Advertising Physical Channel of the LE Advertising Physical Link. Advertising Broadcast (ADVB) is in place for the Logical Transport where the Logical Link is either an ADVB-C for control data or an ADVB-U for user data.

Table 2: Summary of the address types in the Link Layer with their most significant bits (MSB) below their names. Source [25, p. 22 - 24].

Type		Description
Public Device Address		Those addresses are allocated by the Institue of Electrical and Electronics (IEEE)
Random Device Address		Addresses which are randomly allocated. There exist three types. Static, private resolvable and private non-resolvable.
Static (MSB = 0b11)		A randomly generated address. A device is allowed to generate a new Address every time it is power-cycled, but not at any other point in time. It does not disguise the privacy of a device.
Private	Resolvable (MSB = 0b01)	A resolvable private address (RPA) changes periodically. An interval of 15 min is recommended by the Bluetooth Core Specification. However, recommended interval is not a requirement and can be implemented differently. Creating an RPA involves a security key called Identity Resolution Key (IRK). This key is exchanged between devices when they are bonded and is an application of a hash function. A bonded device may resolve an RPA by applying the same hash function with each of the IRK values it possesses from the devices it has bonded with. When this yields a match, the peer knows it has resolved the address and the true identity of the remote device. Not bonded devices cannot resolve the RPA.
	Non-Resolvable (MSB = 0b00)	This is a randomly generated address which changes with every reconnection. It offers privacy protection of the device, without the costs of processing resolvable privacy addresses.

LE Advertising Broadcast

LE Advertising Broadcast (ADVB) or simply advertising provides a connectionless communication mode to transfer data or indicate the presence of a device to a client. Advertisements can be received by anyone who is scanning in the transmitting range. In a one-to-many topology, an advertising device can communicate with many other scanning devices at the same time. Data communication happens only in one direction from the advertising device to the scanning device. A scanning device may respond with a request for further information by forming a connection. However, when a scanning device does this, it is understood by the term Active Scanning. In this work, however, only Passive Scanning is of interest. In this case, a scanning device only listens to and analyzes advertising messages. Furthermore, ADVB is not regarded as a reliable connection because no acknowledgements from the scanning device to the advertising device are sent. The Bluetooth Core Specification defines two types of advertising. Legacy advertising and extended advertising are discussed in more detail in the following.

Legacy Advertising. The advertisements are broadcasted via channels 37, 38 and 39. Those are the so-called Primary Advertising Channels. Only one channel at a time is broadcasting in an arbitrary sequence. The advertisement interval is between 20 ms and 10.24 s [27]. The PDU type is `ADV_IND`. Those packets are 37 octets long with a header of 6 and a payload of at most 31 octets. Each of the Primary Advertising Channels transmits the same packet. Figure 4 shows this schematically.

To avoid persistent packet collisions the transmission of advertisements is delayed by a value known as `advDelay`. It is a pseudo-random value of 0 - 10 ms after each advertising interval. This way simultaneously transmitted advertisements become shifted in time after one interval and permanent collisions are less likely. The `advDelay` is depicted in Figure

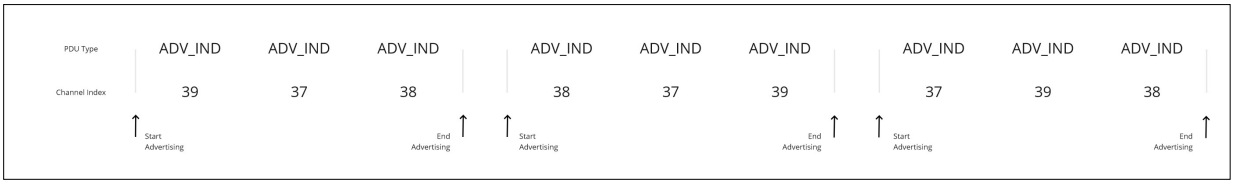

Figure 4: Legacy Advertising using Primary Advertising Channels.
Adapted from [25, p. 32].

5. Legacy Advertising uses a handful of PDU Types. It can be used for undirected or directed advertising. Undirected advertisements can be received by anyone and directed advertisements are meant for a specific device. The PDU Type also stores the information if active scanning is allowed, i.e. whether the scanning device is allowed to establish a connection to the advertising device or not.

Extended Advertising. With Bluetooth Core Specification Version 5 eight new PDU Types were defined. Those additional PDU Types bring a new set of advertising capabilities which turn out to be Extended Advertising. Mostly, it allows larger data with the help of the so-called Secondary Advertising Channels. Those are channels 0 to 36, which carry most of the data. In Extended Advertising, the Primary Channels are used to reference the payload which is transmitted with the Secondary Channels. This allows packets up to 255 octets long. As mentioned before, only header data is sent in the Primary Channels. This includes the so-called Auxiliary Pointer (AuxPtr). It references an auxiliary packet in case more data is transmitted on an additional Secondary Channel. In a nutshell, Extended Advertising enables higher data rates and it operates on all PHYs.

Table 3 lists all PDU Types which are of interest for this work. To recall, Passive Scanning does not rely on a connection to a server, respectively to a tracking device. However, connectable PDU Types might be of importance, because they show the presence of a device. Scannable and non-scannable PDU Types are included as well. Scannable PDU types allow for a scan request to get more advertising data. However, in this work, a scan request is never executed. To summarize, undirected PDUs transmitted by the peripheral, respectively server for Legacy and Extended Advertising are of potential interest.

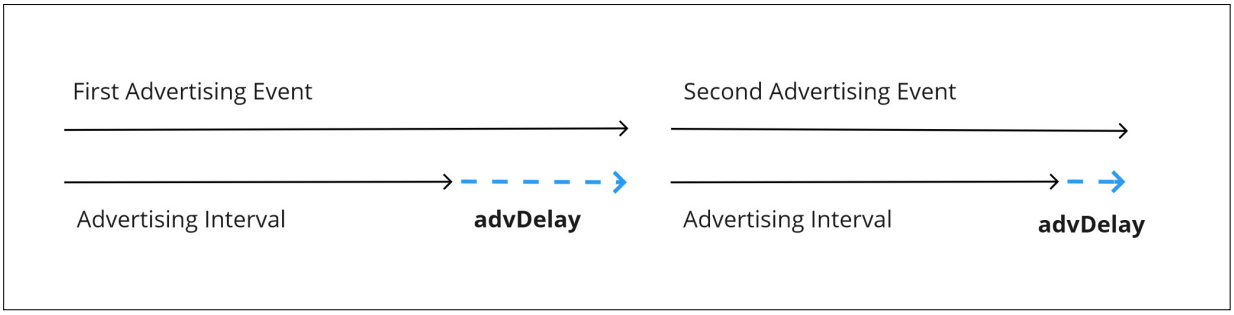

Figure 5: Avoiding permanent packet collision using advDelay. Adapted from [25, p. 33].

Table 3: Summary of all PDU Types which are relevant for this work. Adapted from [25, p. 34]

	PDU Name	Description	Channels	PHY(s)	Scannable	Connectable
Legacy Advertising	ADV_IND	Undirected advertising	Primary	LE 1M	Yes	Yes
	ADV_NONCONN_IND	Undirect, non-connectable, non-scannable advertising	Primary	LE 1M	No	No
	ADV_SCAN_IND	Undirected, scannable advertising	Primary	LE 1M	Yes	No
Extended Advertising	ADV_EXT_IND	Extended advertising	Primary	LE 1M LE Coded	No	Yes
	AUX_ADV_IND	Subordinate extended advertising	Secondary	LE 1M LE 2M LE Coded	No	Yes

3.1.3 Manufacturer Specific Data

Based on [21], the structure of advertisements carrying manufacturer-specific data is explained. The manufacturer-specific data is of importance to identify an AirTag which is explained in more detail in Section 4.5.1.

Advertisements are structured as seen in Figure 2. The payload of an advertisement with manufacturer-specific data is of PDU type `ADV_IND`. Such an indirected advertisement can be received by anyone. The structure of its payload is shown in Figure 6 a). Its AD type defines the content of the payload in the AD Data. Is this set to manufacturer-specific data, the resulting format of its structure is shown in Figure 6 b). The Manufacturer ID is the placeholder for the company identifier. Finally, the Manufacturer Data is defined by the manufacturer itself to send data to other devices. This data can be of help to identify a device, as shown in [24] where Heinrich et al. used this data to identify an AirTag.

3.2 Find My Network

This section explains the Apple Find My network schematically and how it helps to find lost devices based on the work of Heinrich et al. [19]. It is assumed, that offline finding networks from other manufacturers work in a similar manner. Figure 7 gives an overview of how an owner, a lost device, finder devices and Apple's server work together to locate a lost item. First, the owner needs to pair his or her tracking device with an iPhone or mac. Secondly, in case a tracking device gets lost, it broadcasts BLE advertisements that contain a rolling public key. As a third step, iPhones participating in the Find My network which pass by the lost tracking device fetch those BLE advertisements. With the included public key, the finder device uploads an encrypted report to Apple's server. This report includes the location of the finder device as an indication of where the lost device has been found. Lastly, the owner can retrieve the location reports for his or her

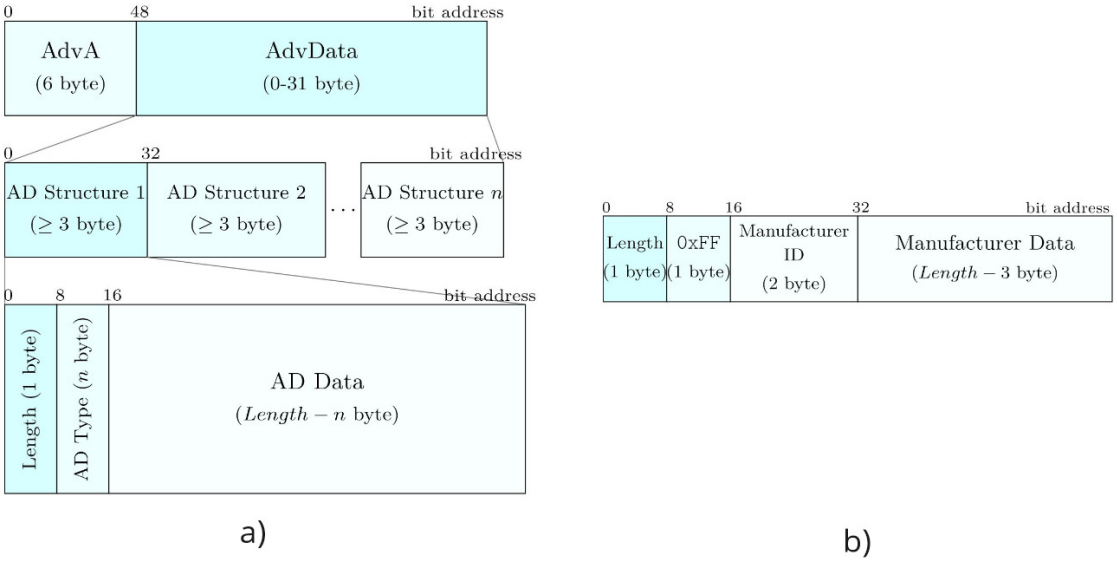

Figure 6: a) shows the format of an indirected advertisement. It contains the advertising address and the advertising data. The advertising data can be subdivided into one or more advertising structures. b) shows the format of manufacturer-specific data. The AD type, set in the AD structure, for manufacturer-specific data is 0xFF. Source [21, p. 53 and 54].

lost device in order to find it. In the following step `losing` is explained in more detail. This is the relevant step in detecting a lost Apple device and therefore of importance for this work. The remaining steps included in the Find My network are explained in the work of Heinrich et al. [19] in more detail, but are not of importance for this work and therefore neglected.

3.2.1 Losing

Heinrich et al. [19] explain in their work the behaviour of a lost Apple device such as an iPhone or a mac for example. AirGuard [24] on the other hand focuses on finding AirTags via the Find My network. Locating both types of devices works the same. It starts with an apple device which is lost and has no internet connection. Such a device broadcasts BLE advertisements. The whole advertisement format is depicted in Figure 8. In its advertisement, it emits the public part of the so-called advertisement key which is 28 bytes long. A BLE advertisement is 37 octets respectively bytes long with a header of 6 and a payload of at most 31 bytes [25]. To be standard compliant with the BLE format for manufacturer-specific data, the advertisement needs 4 bytes for the manufacturer-specific data. This leaves 27 bytes for the payload. In this payload, Apple declares service subtypes such as AirDrop [28] or Find My network [24]. This leaves 25 bytes for the public part of the key which is however 28 bytes long. Apple solved this with the help of the random address field to store the 28-byte long public key [19]. Those public keys are used in the advertisements for 15 min as this is the recommended interval [29]. Afterwards, a

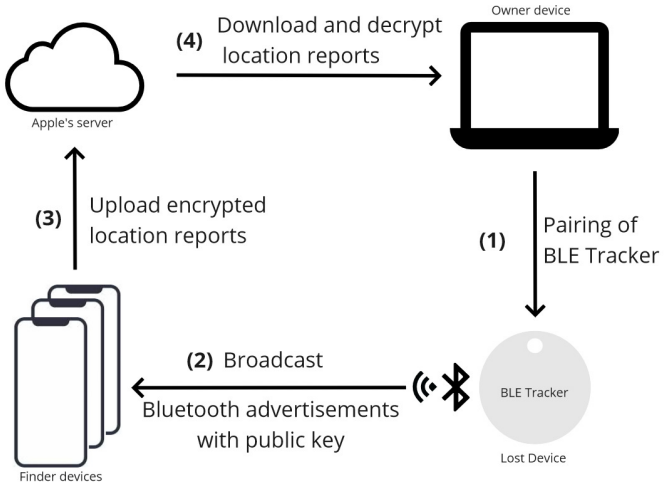

Figure 7: Simplified Findy My network. Adapted from [19, p. 229].

new public key will be created and emitted. Overall, BLE advertisements are emitted in an interval every two seconds once they lose internet connection.

3.3 Tracking Protection

Besides Apple's and AirGuard's tracking protection, other apps exist which protect users from stalking attacks. For example, the Tile and the Tracker Detect app from Apple allow a user to manually scan for malicious trackers. However, this work and the app implemented in parallel focus on scanning for malicious trackers in the background. Since at this point, only Apple's and AirGuard's tracking protection is known to be working in the background, those apps, respectively their tracking algorithms are explained in the following subsections.

3.3.1 Apple's Tracking Protection on iOS

Heinrich et al. [24] reverse-engineered Apple's tracking protection, which is explained here, based on their findings. They identified three tracking algorithms from Apple which detect malicious devices. To avoid tracking, Apple stores all detected devices.

General Detection. Their first algorithm looks at all detected devices in an interval of every two to five minutes. The tracking algorithm classifies all BLE advertisements and from this decides whether a device is malicious or not. The device needs to follow the user for at least 10 minutes and over a distance of 840 meters. The algorithm only considers advertisements received within the last 15 minutes. If a malicious device has been found, the user receives a notification.

Visit-based detection. Apple uses `CLVisit`[3] objects to store a location of a user during

[3] `https://developer.apple.com/documentation/corelocation/clvisit`

Bytes	Content
0–5	BLE address $((p_i[0] \mid (0b11 \ll 6)) \mid\mid p_i[1..5])$
6	Payload length in bytes (30)
7	Advertisement type (0xFF for manufacturer-specific data)
8–9	Company ID (0x004C)
10	Find My network type (0x12)
11	Data length in bytes (25)
12	Status (e.g. battery level)
13–34	Public key bytes $p_i[6..27]$
35	Public key bits $p_i[0] \gg 6$
36	Hint (0x00 on iOS reports)

Figure 8: Find My network format for BLE advertisements. Adapted from [19, p. 232].

a specific period of time. How long it takes for a user to remain at a certain location is, according to Heinrich et al., not specified by apple. Nevertheless, those `CLVisit` objects contain information about when a user has arrived and left a certain location. Those locations in combination with the scanned BLE advertisements at the corresponding locations are used for their visit-based detection algorithms.

Apple's first visit-based algorithm considers all devices of the current and the last visit location. For those locations, all intersecting devices which have been recorded, respectively BLE advertisements which have been received, are potentially malicious. The second algorithm adds some extra checks to those intersecting devices. The algorithm checks for every device if the last advertisement has been received within the last 5 min. It checks whether the device has been travelling with the user for more than 420 m. Finally, if the device has additionally been travelling with the user for the last 5 min the user gets notified. Those thresholds are halved compared to the general detection algorithm as observed by Heinrich et al. [24].

3.3.2 AirGuard's Tracking Protection

AirGuard [24] scans for BLE advertisements in the background. The app uses the Android's `WorkManager`[4] to perform the tracking detection. Those tasks can only be executed every 15 min. AirGuard filters for Find My advertisements and stores those advertisements together with the current geolocation to the database. After a scan, the app runs the detection algorithm. It iterates over all devices based on their BLE MAC address. This address is dependent on the public key used in the advertisements. So for every device, respectively MAC address their algorithm checks if the advertisements of each MAC address have been received within the last 30 minutes. Furthermore, it checks if this device has been fetched at least three times. Lastly, it checks if the device has been near the target user for 400 m depending on the stored geolocations belonging to the corresponding BLE advertisements. Their algorithm is based on the BLE MAC address which depends on the public key. Those keys change once a day at 4 a.m. [30].

[4] `https://developer.android.com/topic/libraries/architecture/workmanager`

Figure 9: Images of all trackers used in this work. a) is an AirTag, b) a Chipolo ONE spot, c) a Galaxy SmartTag+, and d) a Tile tracker.

3.4 Tracking Devices

This section gives an overview of the BLE trackers used in this work. Each subsection explains the features of the corresponding trackers and in which environment they can be used. Figure 9 shows an image of every tracker.

3.4.1 AirTag

The following explanation is based on the official website of Apple [31]. An AirTag is Apple's tracking device for lost items. It has a diameter of 31.9 mm and a height of 8.0 mm. It weighs 11 grams. An AirTag can be attached to keys for example. Afterwards, with the Find My app the owner of the AirTag knows at any point in time where his or her keys are. AirTags have built-in speakers such that the owner can play a sound to find the item more easily. With the ultra-wideband (UWB) technology, an AirTag can be found with so-called Precision Finding. This mimics a compass on the owner's iPhone which guides them to their lost AirTag, respectively item. With the help of BLE, the AirTag advertises that it is lost to other devices in the Find My network. Those devices report the location of the AirTag. In case the AirTag has been set into the Lost Mode by the owner, the finder can get access to the owner's contact information via NFC technology. Only the owner knows where his or her AirTag is. The location is not accessible to Apple. This enables unwanted tracking. However, if a malicious AirTag is near a person, the iPhone of that person informs him or her that a malicious AirTag is close. If this particular AirTag follows that person for a longer period of time, the AirTag will play a sound.

3.4.2 Chipolo ONE Spot

This section is based on the official website of Chipolo [32]. Chipolo works in two ways. Their Chipolo ONE Spot products can be used within Apple's Find My network. This means a Chipolo ONE Spot can be added to the Find My app on an iPhone. Afterwards, it works conceptually like an AirTag. How finding an AirTag, respectively a Chipolo ONE Spot works is explained in section 3.4.1. However, using a Chipolo ONE Spot with the Chipolo app is not possible. The regular Chipolo ONE on the other hand works with the Chipolo app. It allows playing a sound to locate the tracker if it is lost nearby. If the tracker is lost and is not nearby, the app helps to locate the tracker on the map. It shows the tracker's last known location. Those locations are uploaded by other users of the Chipolo app when they pass by the lost item. In this work, only the Chipolo ONE Spot is examined. It can play a sound up to 120 dB within a range of 60 m. It is 37,9 x 6,4 mm small and splashproof. The battery lasts for about one year and can be replaced.

3.4.3 Samsung Galaxy SmartTag+

This section is based on the official description of the Galaxy SmartTag+ from the Samsung website [33]. The purpose of a Galaxy SmartTag+ is to attach it to an item and find it once it's lost. It uses Bluetooth 5.0 and UWB technology. Its battery life is around 165 days. The SmartTag+ has a width and height of nearly 50 mm and a depth of nearly 10 mm. It weighs 14 g. If the attached item gets lost, the owner can figure out where he or she left the item with the help of the Galaxy SmartTag+. If the item is lost nearby, the user has two options to find it. First, Samsung provides users with AR finding. Inside the SmartThings-App, the user is guided to the SmartTag+ with an AR-guided compass. However, this feature is only provided if the user has a Samsung account and is registered at SmartThings[5]. Furthermore, the tag must be paired with a Galaxy Smartphone which runs on Android 11 or higher and is equipped with UWB technology. This is the case for the Galaxy S21 Ultra, Galaxy S21+, Galaxy Note20 Ultra and Galaxy Z Fold2. Secondly, besides AR finding, the user can play a sound on the SmartTag+ via the smartphone. Having a direct line of sight to the SmartTag+, it is supposed to have a Bluetooth range of up to 120 m. Has the device been lost far away, the SmartTag+, respectively item can be located using the Galaxy Find network. Only Galaxy smartphones and tablets build up this network. Further, owners or users of those devices need to agree to participate in the Galaxy Find network. Otherwise, a device would not update the location of the SmartTag+. Besides tracking an item, a SmartTag+ can also be used to control other objects which fall into the SmartThings category. For example, a light can be turned on or off with a SmartTag+.

3.4.4 Tile

The following overview of the Tile is based on their official website [34]. Like all the other trackers, a Tile tracker is meant to be attached to an item and find the item once lost. Tile

[5] `https://www.samsung.com/ch/apps/smartthings/`

provides 4 different trackers. That are the Tile Pro, Tile Mate, Tile Slim, and Tile Sticker [35]. However, they differ only in their appearance but work the same. Nevertheless, this work focuses only on the Tile Pro tracker. It has a Bluetooth range of up to 120 m, runs with a replaceable battery, is water-resistant (IP67) and has a dimension of 58 x 32 x 7.5 mm [36]. Tile trackers need to be paired with a smartphone using the Tile app. It is available for Android and Apple smartphones. Once an item cannot be found and is nearby, the app allows playing a ringing sound on the Tile tracker. This could be further controlled using Amazon Alexa or Google Assistant. Playing a sound works in both ways. Thus, a Tile tracker can be double-pressed to find the missing smartphone. It will also play a sound even when the phone is on silent. Has the item, respectively the Tile tracker been lost far away, the app shows the most recent location of the tracker. The Tile network picks up signals from Tile trackers and updates the most recent location. The Tile network is composed of all the smartphones running the Tile app and network extenders. Furthermore, the Tile tracker has a QR code engraved. Within the app, the contact information can be stored which a person receives by scanning the QR code. Furthermore, the Tile network is open to adding devices from partners such as HP, Fitbit, Skullcandy, Dell and many more. After a 30-day free trial, a Tile user however needs to update to Tile Premium to use the tracking service. It costs 3 CHF on a monthly base and 32 CHF on an annual base.

Chapter 4

Design and Implementation: Android App

Parallel to this thesis an Android app has been developed[6]. This section describes its configurations, schematic workflow and architecture. The capability of Android's BLE API is introduced and eventually how the trackers used in this work are identified. This section not only explains the app but also serves as documentation for future developers.

4.1 Build Configurations

Android Studio manages the build process with Gradle[7]. The resources, source code, and packages as well as any dependencies of an app are converted into an Android Application Package (APK). This compilation process is managed by Gradle. Those APK files are needed to test, deploy and distribute an Android app. For this project, Gradle 7.3.3 and the Android Gradle Plugin 7.2.1 are in use.

In the build.gradle file the `compileSDK = 32`, `minSDK = 26` and `targetSDK = 28` are declared. The most important one is the `minSDK`. It determines the minimum OS version

[6] `https://github.com/LouisBienz/HomeScout`
[7] `https://gradle.org/`

Android Platform Version	7.0 Nougat	**8.0 Oreo**	8.1 Oreo	9.0 Pie	10
API Level	25	**26**	27	28	29
Cumulative Distribution	90.4 %	**88.2 %**	85.2 %	77.3 %	62.8 %

Figure 10: Android API Version Distribution accessed on 26.09.2022 of the Help Me Chose functionality of the Create Project Wizard. Adapted from [37, Figure 3].

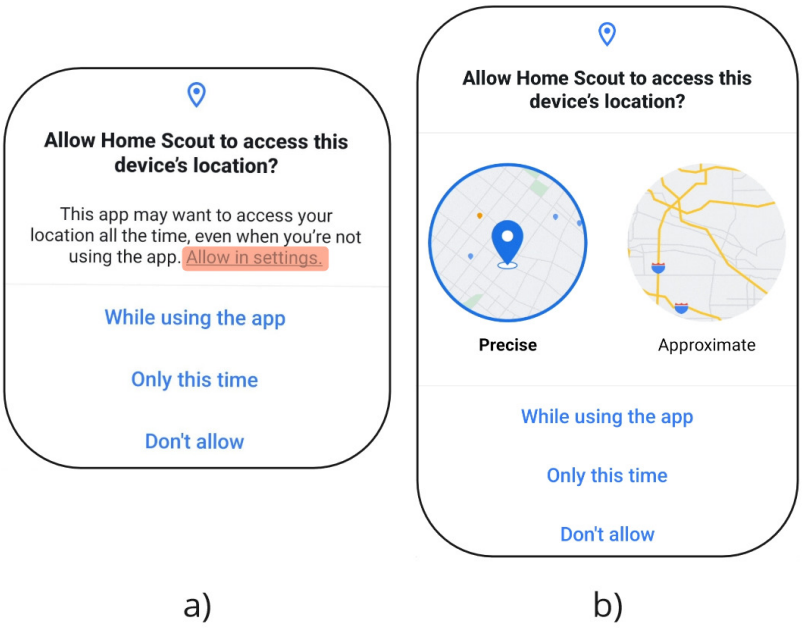

Figure 11: a) shows the system message for SDK version 28. The red-marked area allows the user to jump into the settings where he or she can allow to use the location permission ”all the time”. b) shows the system message for SDK version 32. It is not possible to allow the locations ”all the time” directly. The user would have to do this manually via the app's settings.

for the app. Devices with a version below 26 are not capable to run the app. `minSDK = 26` or Android Oreo (8.0) introduced the support for the BLE 5.0 standard [38]. Since this app relies on those BLE capabilities, supporting any lower APIs would not make sense. Furthermore, the `Create New Project Wizard`[8] provides a cumulative distribution of the Android Platform Version. With the `Help Me Choose` functionality, it provides figures about how many devices use which Android version, respectively on how many devices the app will run with the chosen API. Figure 10 represents those numbers in a shortened and adapted version. As expected, the higher the API version is, the fewer devices are supported. To make the app available to as many users as possible it makes sense to set the minSDK to the API Level 26, as nearly 90% of the devices will be capable to run this app. Besides the `minSDK` the `targedSDK` defines for example how system messages appear. Those system messages of `targetSDK = 28` fit best the use case of this app. The reason is, that the app needs to have the permissions `ACCESS_BACKGROUND_LOCATION` and `ACCESS_FINE_LOCATION`[9]. The system message of the `targetSDK = 28` allows the user to switch to the permission settings directly. Therefore, the user can grant all the aforementioned permissions accordingly. Higher SDKs on the other hand restrict the user to enable the `ACCESS_BACKGROUND_LOCATION` directly via the system message as shown in Figure 11. Because it is inevitable for this app to use the `ACCESS_BACKGROUND_LOCATION` permission, the `targetSDK` is set to 28. Using this version, a user can directly navigate to the appropriate permission settings, without doing a manual detour.

[8] `https://developer.android.com/studio/projects/create-project`
[9] `https://developer.android.com/training/location/permissions`

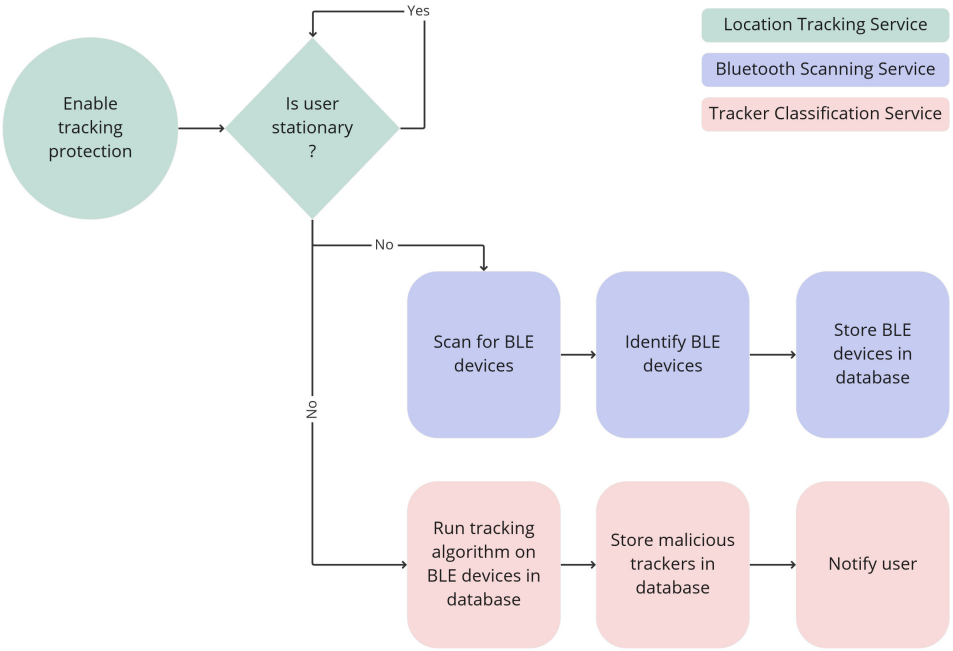

Figure 12: Sketch of the app workflow. Once the user enables tracking protection, the app regularly evaluates if the user is stationary. If the user starts moving, the app starts scanning for BLE devices and stores them in the database. Concurrently, the tracking algorithm classifies the devices into malicious and non-malicious trackers. If a malicious tracker is found, the user gets notified.

4.2 App Workflow

The main functionality of this Android app is to scan, identify and classify BLE trackers as malicious or non-malicious trackers. This is divided into three services. How those three services interact conceptually is shown in Figure 12. Firstly, tracking protection needs to be enabled. Afterwards, the app evaluates if the user is moving or stationary. Once a user starts to move, BLE advertisements are scanned and stored in a database. In parallel, the tracking algorithm tries to detect any malicious BLE device among all scanned BLE devices, respectively BLE advertisements. All those Services are explained in more detail in Section 4.3.3.

4.3 App Architecture

This app follows the Developer Guides[10] from Android and their best practices. The general architectural components of the app are the User Interface (UI), the data layer and the services. In the following, those components are described in more detail. Figure 13 shows the general architecture of the app. This helps as a starting point for possible subsequent developers.

[10] `https://developer.android.com/guide`

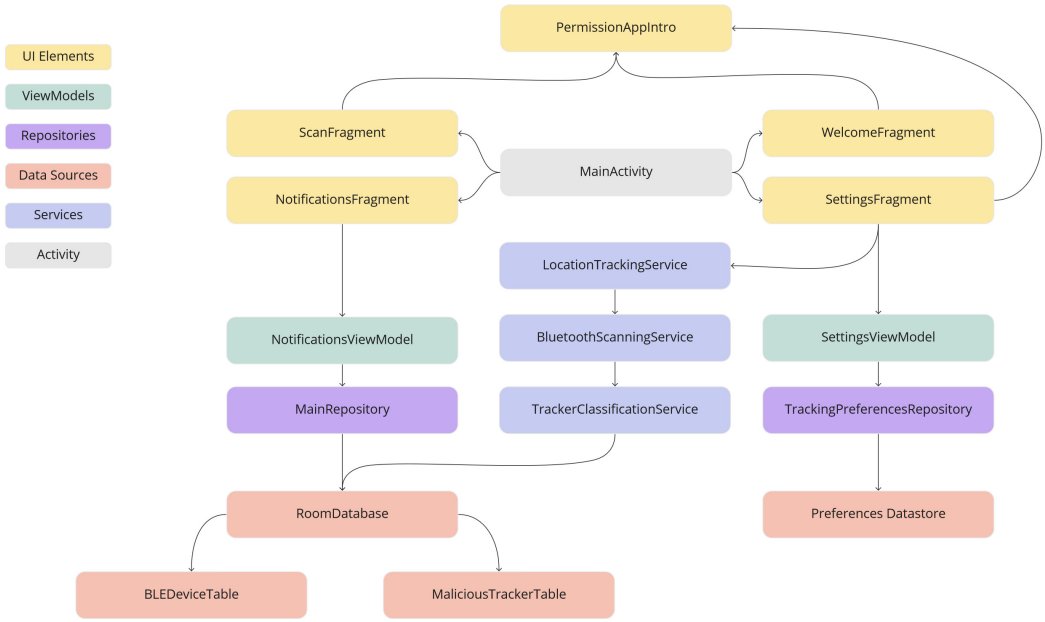

Figure 13: Visualizing the most important classes used in this project sketched with some of their relations. This serves as documentation for the app as well as for subsequent developers to quickly get on overview of the code base.

4.3.1 User Interface (UI)

The UI is subdivided into two parts. The app intro and the UI layer itself. The app intro consists of four screens used to request the necessary permissions. The UI layer also counts four screens which allow the user to interact with the app. Both parts are explained in more detail.

App Intro

The app intro is used to guide the user in a user-friendly manner through the permissions to be granted which are necessary for the app. The AppIntro Library[11] has been used for this task. This library provides an easy way to display slides in a carousel as well as enables the app to request the necessary permissions. In case those permissions are not granted and the user wants to execute a function of the app which requires those permissions, the app intro gets launched. It consists of four screens. The first screen informs the user about the Bluetooth permission and provides the dialog to grant them. The second screen informs the user about the location permissions and that the app needs to access background locations in order to be able to detect malicious trackers properly. The third screen is a placeholder for the request to ignore battery optimization. At this point, the proper permission request according to Android's developer Guide has not been implemented. This permission is needed such that the app is able to run even when the user locks the phone, respectively when the phone is in doze mode. This needs to be

[11] `https://github.com/AppIntro/AppIntro`

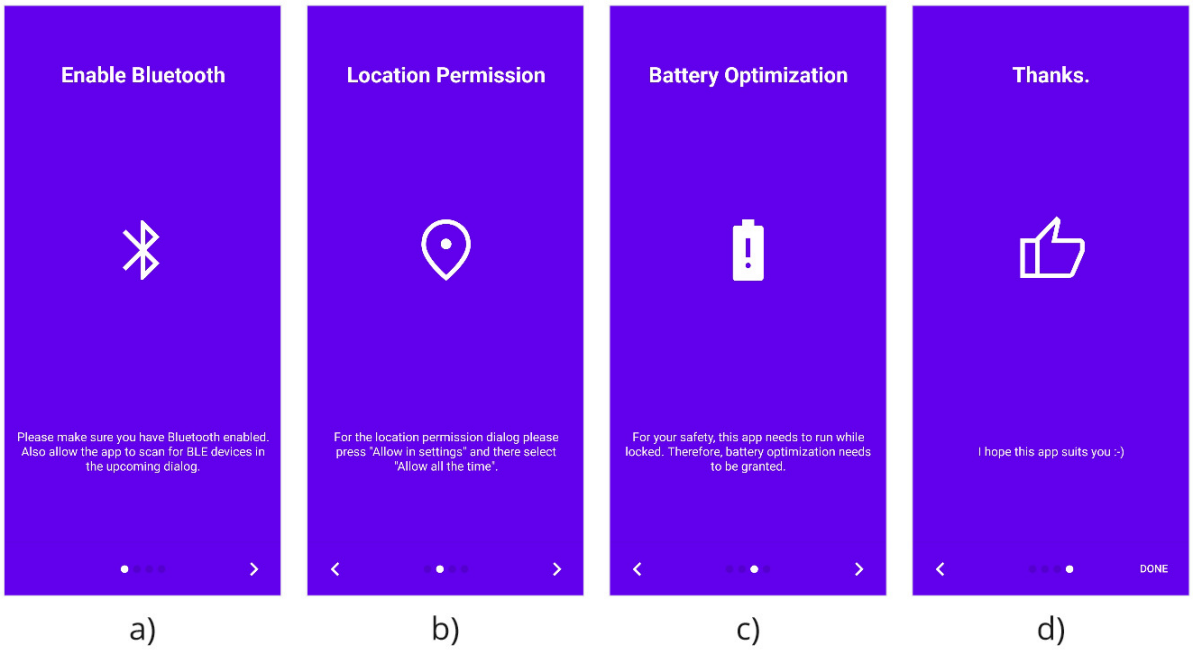

Figure 14: Screenshots of the four slides in the app intro. Moving from one slide to the next one invokes the system services to grant the corresponding permissions. An example of such a system service asking for location permission is depicted in Figure 11. a) asks for Bluetooth, b) for location, and c) for battery optimization permissions. Finally, d) thanks the user for granting the permissions.

implemented in the next version and is therefore noted in Chapter 7. Nonetheless, the app could still be used to test its functionality and analyse its tracking algorithm even without this permission. The last screen simply thanks the user for granting the requested permissions. Screenshots of those four screens can be seen in Figure 14.

UI Layer

The UI Layer is built according to Android's developer Guide which implicitly implements the Model View ViewModel (MVVM) pattern[12]. The UI Layer consists of a MainActivity which uses a bottom navigation bar to let the user navigate between the fragments. The MainActivity hosts the Welcome, Notifications, Settings, and Scan fragments. Figure 15 shows screenshots of those four fragments including the bottom navigation bar. The Welcome fragment gives a short introduction to the purpose of this app and why the permissions are required. The Notifications fragment displays all malicious trackers detected by the app's tracking protection. The tracking protection can be enabled on the Settings fragment. It also allows setting the preferences of the tracking algorithm. Furthermore, three buttons provide default preferences. The ones evaluated within this work which are the default preferences, the ones AirGuard uses and finally the ones Apple uses on the iPhone to detect AirTags. The Scan fragment allows users to manually scan for any BLE device. This fragment does not provide much functionality in terms of stalking protection

[12] `https://www.geeksforgeeks.org/mvvm-model-view-viewmodel-architecture-pattern-in-android/`

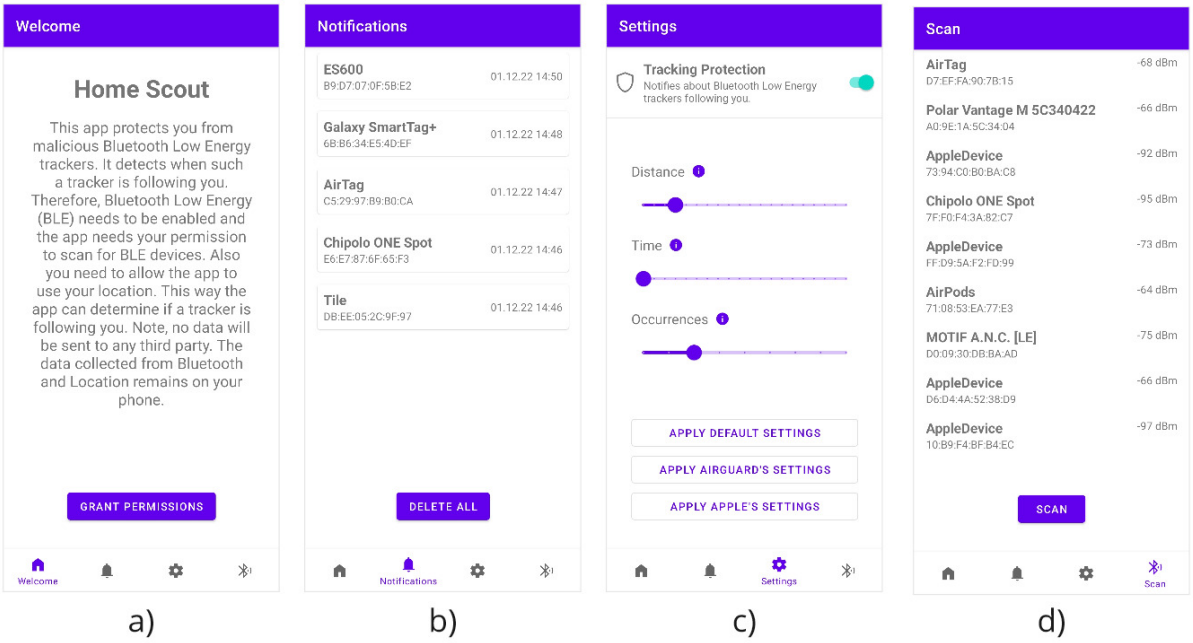

Figure 15: Screenshots of the four screens which represent the main features of the app. a) is the Welcome, b) the Notifications, c) the Settings, and d) the Scan Fragment.

for the user. It was however needed for the experiments regarding the tracking devices as explained in Section 5.1.1 and the RSSI values described in Section 5.1.2.

As the MVVM pattern suggests, it separates and defines the communication between a Model, View and a ViewModel. The Model is responsible for the data sources. This is explained in more detail in the Subsection 4.3.2. For the UI layer, the View respectively UI elements and the ViewModel are relevant. Figure 16 a) illustrates how these two components interact with each other. The View is responsible to handle user interactions and displays data to the user by observing the ViewModel. In this app, only the Notifications and Settings fragment rely on data. Therefore, the corresponding NotificationsViewModel and SettingsViewModel are in place as seen in Figure 13. The Welcome and the Scan fragment do not rely on data as it is not needed. Hence, no WelcomeViewModel and ScanViewModel are implemented.

4.3.2 Data Layer

The data layer consists of repositories and data sources as seen in Figure 16 b). A repository is used to access and gather data from the data sources and make this data available to the UI layer. Within the architecture of this app, two repositories are needed. The MainRepository and the TrackingPreferencesRepository. The MainRepository is the interface for interactions with a RoomDatabase. The TrackingPreferencesRepository is the interface for the Preferences DataStore. Both are explained in more detail in the following.

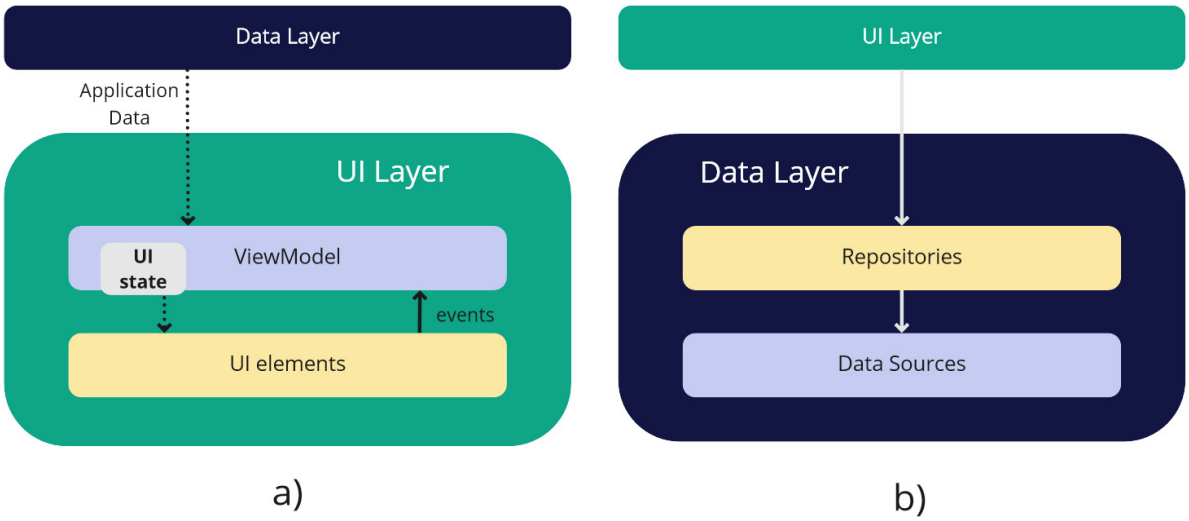

Figure 16: a) Diagram of the UI layer within an MVVM pattern. The data layer, ViewModel, and UI elements in this figure correspond to the Model, the ViewModel, and the View of the MVVM pattern. Adapted from [39]. b) Diagram of the data layer within an MVVM pattern. The data layer corresponds to the Model of the MVVM pattern. Adapted from [40].

Room Database

The Room Database has been implemented according to Android's developer Guide[13]. In this project, two data entities are stored in the database. One for the scanned BLE devices which are stored in the `ble_device_table` and one for the BLE devices which have been classified as malicious, which are stored in the `malicious_tracker_table`. The schema for both tables is depicted in Figure 17. Furthermore, for every data table, a Data Access Object (DAO) has been implemented. The `BLEDeviceDao` is responsible for SQL queries on the `ble_device_table` and the `MaliciousTrackerDao` is responsible for SQL queries on the `malicious_tracker_table`. The DAO build the Data Source according to Figure 16. The task of the repository is to abstract the data from the rest of the app [41]. In this case, it abstracts the data from the UI layer. For this app, the MainRepository is used to bundle all operations on the database. Hence, it operates using the instances of the `BLEDeviceDao` and the `MaliciousTrackerDao`. Since both DAOs only provide very basic SQL queries, it has been decided to provide both of them in the MainRepository. In case the app needs more complex operations on the database, which means that the DAOs will grow in functionality, it is recommended to split the MainRepository into a BLEDeviceRepository and a MaliciousTrackerRepository.

[13] `https://developer.android.com/training/data-storage/room`

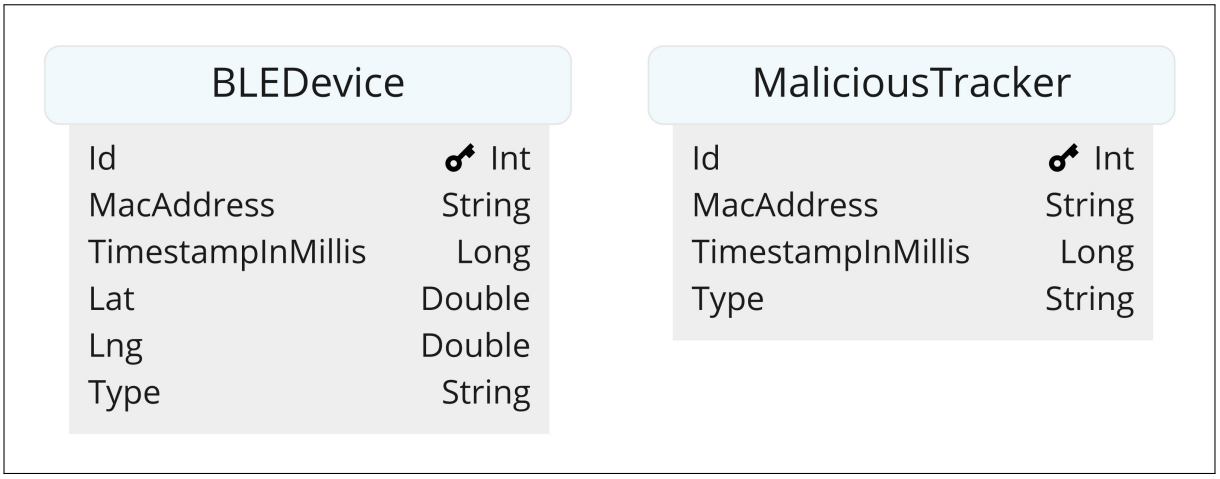

Figure 17: Database schema for all data tables used in this work. On the left, the database schema for BLE devices is shown. On the right, the schema for MaliciousTrackers is depicted. The key icon states which attribute represents the primary key of a table.

Preferences DataStore

The Preferences DataStore has been implemented according to Android's developer Guide[14]. It is a solution to store key-value pairs. This perfectly fits to store the tracking preferences a user can define on the Settings fragment as seen in Figure 15. The read and write operations to the DataStore are defined in the `TrackingPreferencesRepository`. Those are simple operations. Therefore, the data source and the repository according to Figure 16 are combined within the `TrackingPreferencesRepository`.

4.3.3 Services

The services are implemented according to Android's developer Guide[15]. The whole functionality representing the tracking protection is implemented in three different services. The first service is responsible to track the location of the user and the second service scans for BLE devices if needed. The third service runs a classification algorithm on the scanned BLE devices and informs the user once a malicious tracker has been detected. The individual services are explained in more detail in the corresponding sections below.

All services are so-called Foreground Services. This means, once the user enables the tracking protection on the Settings fragment as seen in Figure 15 c), the services do their work while the user can still interact with the app. Furthermore, the app can be closed and the services still run. If the user locks his or her phone, the phone switches to the doze mode and the operations of the services are only executed in the maintenance windows [42]. This breaks the tracking protection to run properly. However, apps that keep a user safe, are allowed to still run while the phone is in doze mode [42]. For this however an

[14] `https://developer.android.com/topic/libraries/architecture/datastore`
[15] `https://developer.android.com/guide/components/services`

ACTION_REQUEST_IGNORE_BATTERY_OPTIMIZATIONS needs to be implemented in the app [43], which could not yet be implemented properly and is therefore not included in the app so far. Nonetheless, the tracking protection works as expected while the phone is not locked and the experiments could be executed as needed. Finally, once the user disables the tracking protection, the services stop and free their resources.

Location Tracking

The idea behind the LocationTrackingService is to permanently evaluate if a user is stationary or moving. Once the user is moving, the BluetoothScanningService starts scanning for devices and eventually the TrackerClassificationService evaluates if a tracker is following the user or not. The app workflow in Figure 12 illustrates this behaviour. The decision to only scan for BLE devices when the user moves is based on two reasons. Firstly, scanning for BLE devices consumes a lot of battery. Therefore, the app should not execute BLE scans permanently. Secondly, while a user is stationary, it is assumed that there is no threat of being tracked. But as soon as the user leaves a stationary place, the tracking protection should inform the user about malicious trackers. For example, a user at work, respectively being stationary, is at first not of much interest to an adversary. A stationary victim however creates the opportunity for an adversary to slip a tracker into the victim's belongings. Afterwards, as soon as the user leaves his or her stationary place, it becomes of interest to the adversary where the user is heading, respectively the actual stalking attack of a victim begins. At this point in time, it becomes of importance to scan for malicious trackers. Therefore, the LocationTrackingService permanently evaluates if a user is moving. If so, the BluetoothScanningService and the TrackerClassificationService start such that malicious trackers can be detected. Once the user is stationary again, the BluetoothScanningService and TrackerClassificationService stop.

To determine if a user is stationary the LocationTrackingService uses the function shown in Figure 18. Figuratively speaking, the user draws a tail behind him or her. This tail is composed on the basis of the positions where the user has been. It is limited in time to about two minutes. If the length of the tail exceeds a certain distance, it is assumed that the user is moving. Figure 19 conceptually illustrates two examples of potential user tails.

The user's tail is created with a ring buffer (or circular buffer) data structure. Its size is determined to hold a user position tail of approximately 2 minutes. This corresponds to a size of 24 elements. This number is derived from the configuration of the fusedLocationProviderClient, respectively the LocationRequest update interval which is set to 5 seconds. However, the location updates can occur faster or slower [44]. Furthermore, the ring buffer provides the function getElementsOrderedTailToHead(), called in line 177 (Figure 18). It returns the user's position history ordered from tail to head. Looping over those positions until the second last index of the history ring buffer allows computing the distance between the current and consecutive position for every iteration as it is shown in line 192. At the end of the loop, the computed distance is added to an array containing all distances calculated from tail to head. Finally, the function returns true or false in line 196, depending on whether the total distance travelled is smaller or equal to the constant STATIONARY_MOVING_DISTANCE, which is set to 50 meters.

```
174     private fun isUserStationary(): Boolean {
175
176         val allDistancesTraveled = mutableListOf<Float>()
177         val orderedUserPositionBuffer = userPositionsHistoryBuffer.getElementsOrderedTailToHead()
178         val secondLastIndex = orderedUserPositionBuffer.size - 2
179
180         for (i in 0..secondLastIndex) {
181
182             val currentLocation = Location("currentLocation").apply {
183                 latitude = orderedUserPositionBuffer[i].latitude
184                 longitude = orderedUserPositionBuffer[i].longitude
185             }
186
187             val nextLocation = Location("nextLocation").apply {
188                 latitude = orderedUserPositionBuffer[i + 1].latitude
189                 longitude = orderedUserPositionBuffer[i + 1].longitude
190             }
191
192             val distanceBetweenTwoLocations = currentLocation.distanceTo(nextLocation)
193             allDistancesTraveled.add(distanceBetweenTwoLocations)
194         }
195
196         return allDistancesTraveled.sum() <= STATIONARY_MOVING_DISTANCE
197     }
```

Figure 18: Code snippet of the *LocationTrackingService* which decides if a user is stationary or moving.

This implementation which evaluates if the user is stationary or moving might return false results in at least two cases. First, the function evaluates that the user is not moving, even though the user is actually moving. This is the case when a user is travelling very slowly. As mentioned before, the function captures the user's position history of approximately two minutes. Furthermore, within those two minutes, the user has to travel more than 50 meters in order to be considered moving. Hence, if the user walks less than 50 meters in two minutes, which equals to a constant velocity of 1.5 km/h, the user will never be considered moving and the tracking algorithm will never start, respectively detect any malicious trackers. Second, the function evaluates that the user is moving, even though the user is regarded as stationary. This happens if the user's stationary movement distance is larger than 50 meters. To give an example, a user works at a construction site, which is assumed to have a radius of 100 meters. The user needs to regularly move more than 50 meters to do his or her job. In this case, the function would consider the user to be moving, even though the whole construction site is regarded as the stationary place of the user. Finally, it needs to be noted, that once a user comes to a rest, respectively is regarded as transitioning from the moving to the stationary state, it takes the algorithm approximately two minutes to realize because the tail of the user position history needs to catch up with the head, which represents the current stationary position.

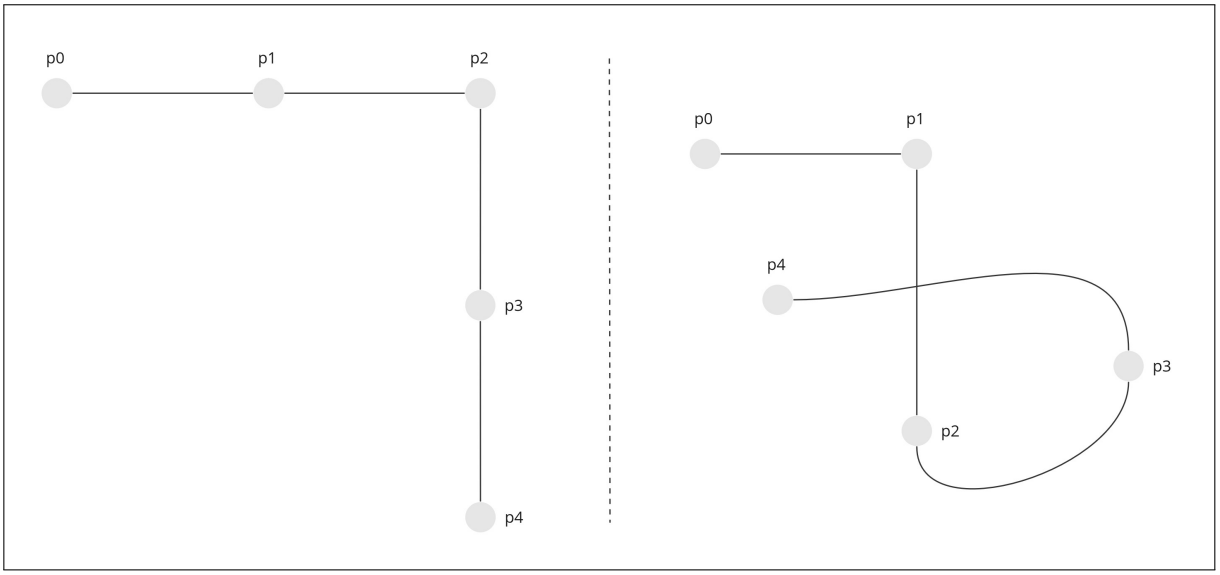

Figure 19: This figure shows two examples of a user tail that is composed on the basis of the visited positions. In both examples, the user starts at position $p0$ and walks up to position $p4$. For those examples, it is given that every edge between two consecutive positions is equidistant and equal to 5 meters. Furthermore, it is given that the threshold which determines if the user is moving is set to 15 meters. Using the function $isUserStationary()$ implemented in the $LocationTrackingService$ evaluates both scenarios as false, respectively that the user is moving, because at $p4$ the user has travelled a distance of 20 meters.

Bluetooth Scanning

The `BluetoothScanningService` handles how the app scans for BLE devices. Figure 20 shows the code snippet of the function `startBleScan`. It is important to note that scanning permanently for BLE devices drains the battery [45]. Therefore, an appropriate scan period and more important a scan interval need to be selected. If the scanning service is not running, scanning for BLE devices is stopped as declared in lines 173 - 176. As long as the `BluetoothScanningService` is running, the function starts to scan for BLE devices as indicated in lines 169 - 171.

Before the scanning starts, a handler defines the scan period and the scan interval. The handler from lines 157 - 167 defines the number of seconds to scan for BLE devices, respectively the scan period. The body of this function is invoked after the constant `SCAN_PERIOD` which is set to 12 seconds. After this period, the scanning is stopped (line 159) and the BLE devices are inserted into the database (line 160). The scan period is set to 12 seconds because the BLE advertisement interval can vary between 20 ms and 10.24 s [27]. Therefore, scanning for 12 seconds makes sure to capture a signal of every BLE device, as in this period every BLE device advertises at least once.

Within the handler regarding the scan period, a second handler is declared in lines 163 - 165, which invokes the function `startBleScan()`. Hence, the function runs in a recursive loop with a delay according to the constant `INTERVAL_BLE_SCAN`. This constant is set to 18 seconds. Therefore, the whole scanning process is executed twice within one minute.

```
154         if (!isScanning && isServiceRunning) {
155
156             // stop scanning after SCAN_PERIOD
157             handler.postDelayed({
158                 isScanning = false
159                 bleScanner.stopScan(scanCallback)
160                 insertScanResultsInDb()
161
162                 // loop: start scanning after INTERVAL_BLE_SCAN
163                 handler.postDelayed({
164                     startBleScan()
165                 }, INTERVAL_BLE_SCAN )
166
167             }, SCAN_PERIOD)
168
169             scanResults.clear()
170             isScanning = true
171             bleScanner.startScan(scanCallback)
172
173         } else {
174             isScanning = false
175             bleScanner.stopScan(scanCallback)
176         }
```

Figure 20: Code snippet of the *BluetoothScanningService* which handles the scan period and scan interval.

Once the user is regarded as stationary or the user disables the tracking protection, the recursive loop stops and the app terminates to scan for BLE devices.

Lastly, the `BluetoothScanningService` is responsible to store the scanned devices in the database. As seen in Figure 17, a BLE device has a `lat` and `lng` attribute which describes the latitude and longitude at which the BLE device has been scanned. Those coordinates are derived from the last known location of the smartphone in use. Hence, those are not the true coordinates of a BLE device, but rather at which point a signal of the BLE device has been captured by the user.

```
169             hashMapBleDevicesSortedByTime.let{ hashMapBleDevicesSortedByTime ->
170
171                 for (key in hashMapBleDevicesSortedByTime.keys) {
172
173                     // get all scans of the same mac address
174                     val scansOfThisDevice = hashMapBleDevicesSortedByTime[key]!!
175
176                     // check if more than one scan exists or it has less scans than defined by user
177                     if ( scansOfThisDevice.size == 1 || scansOfThisDevice.size < occurrences!!) {continue}
178
179
180                     // check if the tracker follows according to time defined by user
181                     val youngestScanTime = scansOfThisDevice.first().timestampInMilliSeconds
182                     val oldestScanTime = scansOfThisDevice.last().timestampInMilliSeconds
183                     val diffBetweenYoungestAndOldestScan = youngestScanTime - oldestScanTime
184                     val timeThresholdInMillis = timeinMin!! * 60000
185                     if (diffBetweenYoungestAndOldestScan < timeThresholdInMillis) { continue }
186
187
188                     // check if the tracker follows according to distance defined by user
189                     var distanceFollowed = 0.0
190                     val secondLastIndex = scansOfThisDevice.size - 2
191                     for (i in 0..secondLastIndex) {
192
193                         val currentLocation = Location("currentLocation").apply {
194                             latitude = scansOfThisDevice[i].lat
195                             longitude = scansOfThisDevice[i].lng
196                         }
197
198                         val nextLocation = Location("nextLocation").apply {
199                             latitude = scansOfThisDevice[i + 1].lat
200                             longitude = scansOfThisDevice[i + 1].lng
201                         }
202
203                         val distanceBetweenTwoLocations = currentLocation.distanceTo(nextLocation)
204                         distanceFollowed += distanceBetweenTwoLocations
205                     }
206
207                     if (distanceFollowed < distance!!) { continue }
```

Figure 21: Code snippet of the *TrackerClassificationService* which decides if a tracker is malicious or non-malicious.

Tracker Classification

The `TrackerClassificationService` is the heart of the app. Figure 21 shows the lines which accomplish this task. The variables `occurrences, timeInMin & distance`, used in lines 177, 184 and 207 are the values provided by the data store, respectively the user and make this tracker classification algorithm adaptable. To save battery this code is executed every 30 seconds. It is assumed that in a period of 30 seconds, all devices in the database are classified.

The `TrackerClassificationService` iterates over all scanned BLE devices which are stored in the `hashMapBleDevicesSortedByTime`. This hashmap contains as key the MAC address of a tracker. The corresponding value is a list of all BLE devices, as declared in the database schema seen in Figure 17, with the same MAC address. The values in the list are sorted in time. The first value is the value with the youngest timestamp, the last

value represents the scan with the oldest timestamp. The algorithm evaluates if a tracker is malicious or not based on the three parameters `occurrences, timeInMin & distance` set by the user.

In line 177, the algorithm first checks the occurrences. At least two occurrences have to exist, otherwise, it is not possible to make any statements regarding the distance and time parameters. Therefore, the statement `scansOfThisDevice.size == 1` is included in the if statement within an OR clause. The second part evaluates if fewer scans exist as defined by the user. If both evaluate to false, the loop continues with the next key, respectively MAC address. Otherwise, the device is not regarded as malicious.

Secondly, the algorithm checks if the time criterion is fulfilled. The difference in time between the youngest and oldest scans is compared with the time set by the user. The comparison happens on the basis of milliseconds, that's why the user parameter `timeInMin` is multiplied with 60'000. If the time difference is smaller than the user-defined value, the loop continues, as seen in line 185. Otherwise, the device is not regarded as malicious.

Lastly, the algorithm checks if the tracker has followed the user for the defined distance parameter. The computation of the distance is done in the same manner as in the `LocationTrackingService`, where the distance between consecutive locations is added and all those distances are summed up. This behaves in the same way as shown in Figure 19. If the summed distances, as seen in line 207, are smaller than the distance value set by the user, the algorithm continues. Also here, the device is not regarded as malicious otherwise.

This sums up the classification algorithm. Hence, if all criteria are met, respectively the code has never continued on any of the described if-statements, the service stores the current BLE device into the database as a malicious tracker, as defined in the database schema shown in Figure 17. Finally, the user gets notified instantly and the malicious tracker is displayed on the Notifications screen as seen in Figure 15 b).

4.4 Android Bluetooth Low Energy API

To scan for BLE signals the app needs Android's `BluetoothLeScanner`[16]. It provides the functionality to start and stop a scan. The `startScan()` method needs a callback[17] which handles the scan results. Since the app is a passive scanner only the classes `ScanResult`, `ScanRecord`, and `BluetoothDevice` are needed. All mentioned components are explained in more detail in the following sections.

4.4.1 BluetoothLeScanner

The `BluetoothLeScanner` in the app is initialized using the lazy property in Kotlin. This way, only one instance of the `BluetoothLeScanner` is created [46]. This makes sure, that

[16] `https://developer.android.com/reference/android/bluetooth/le/BluetoothLeScanner`
[17] `https://developer.android.com/guide/topics/connectivity/bluetooth/find-ble-devices`

always the same and only one instance is scanning for BLE signals. Its `startScan()` is executed with the default parameters. No `ScanFilter` and no `ScanSettings` are given. The default settings are `SCAN_MODE_LOW_POWER`, which consumes the least power [47].

4.4.2 ScanResult, ScanRecord, and BluetoothDevice

A `ScanResult`[18] is returned from the `BluetoothLeScanner` and can be processed in the self-defined callback, handed to the `startScan()` function as a parameter. The `ScanResult` allows to access the `ScanRecord`[19] and `BluetoothDevice`[20]. Example return values for an AirTag, Chipolo ONE Spot, Galaxy SmartTag+, and the Tile tracker are shown in Appendix A. Those three elements build all the accessible information an Android phone is able to receive from BLE advertisements. As this app is a passive scanner, no Bond or GATT connection to any BLE device is established.

Based on this information, it is assessed in this work if a holistic identification of BLE trackers is possible. The result of such a holistic tracker identification would be to classify BLE advertisements into advertisements coming from trackers or non-trackers. This is evaluated in Chapter 5.3. Besides that, the four trackers can be identified as such. Meaning, a BLE advertisement can be broken down such that it can be recognized as a BLE advertisement from an AirTag for example. How every tracker used in this work is identified is described in the following section.

[18] `https://developer.android.com/reference/android/bluetooth/le/ScanResult`
[19] `https://developer.android.com/reference/android/bluetooth/le/ScanRecord`
[20] `https://developer.android.com/reference/android/bluetooth/BluetoothDevice`

Table 4: Identification of Apple's device types from the manufacturer-specific data. Source [24, Table 3].

Device Type	Bits	Category	Example
Other	0b00	Apple devices	iPhone, Mac, iPads
D (Durian)	0b01	AirTags	AirTag
H (Hawkeye	0b10	3rd Party	Chipolo ONE Spot
HELE	0b11	Headphones	AirPods Pro

```
19          val manufacturerData = scanResult.scanRecord?.getManufacturerSpecificData(0x004c)
20          val services = scanResult.scanRecord?.serviceUuids
21          if (manufacturerData != null) {
22              val statusByte: Byte = manufacturerData[2]
23  //            Timber.d("Status byte $statusByte, ${statusByte.toString(2)}")
24              // Get the correct int from the byte
25              val deviceTypeInt = (statusByte.and(0x30).toInt() shr 4)
```

Figure 22: How AirGuard identifies an AirTag. Source [48].

4.5 Tracker Identification

The trackers used in this work can be identified as such from a BLE advertisement. Those identifications on an individual level are explained in the following sections. This enables to notify users which type of tracker is following them, as seen for example in Figure 15 b) on the Notifications fragment.

4.5.1 AirTag

How an AirTag can be identified has been intensively studied by Heinrich et al. [24]. They reverse-engineered the BLE advertisements of Apple devices and were able to identify four different types of devices. One of which is the AirTag. All other device types including the AirTag are listed in Table 4. This identification of device types is based on the manufacturer-specific data that Apple devices broadcast in their advertisements. The interpretation and processing of this manufacturer-specific data is necessary to eventually identify an AirTag as such. AirGuard's code by Heinrich et al. to identify an AirTag has been used in this work. For this reason, their implementation regarding the interpretation of the manufacturer-specific data, as shown in Figure 22 is explained step by step.

Table 5: Apple's advertisement format. Adapted from [24, Table 1]

Bytes	Content
0-5	BLE address
6	Payload length in bytes (30)
7	Advertisement type (0xFF for manufacturer-specific data)
8-9	Company ID (0x0004C)
10	Offline Finding type (0x12)
11	Data length in bytes (25)
12	Status (e.g. battery level)
13-34	Public key bytes
35	Public key bits
36	Hints (0x00 on iOS reports)

First of all, the `ScanRecord` contains the manufacturer-specific data. In order to access Apple's manufacturer specific data, the method `getManufacturerSpecificData(0x004c)` of Android's BLE API is used. Its argument 0x004c represents Apple's company identifier [26]. This function returns null if the BLE advertisement does not contain any manufacturer-specific data or the wrong company identifier has been used. Figure 22, line 19 shows the access of the manufacturer-specific data and line 21 does the null check. Thus, the body of the if statement is only executed on Apple's manufacturer-specific data.

Within line 22, AirGuard retrieves the status byte. The format of this status byte is explained in Table 5. To retrieve one of the four Apple device types, as shown in Table 4, a bitwise `and` operation with a value of 0x30 and a right shift of 4 bits is applied. How this ends up as one of the 4 different bits as seen in Table 4 is explained in more detail.

Firstly, Kotlin's `.and()` and `shr` functions are explained. The following explanations of the `.and()` and `shr` operations are based on an online example [49]. The `.and()` operation is a bitwise operation. This means, the actual `.and()` is applied on the caller's binary representation. The argument as well will be shown in binary representation in the examples below. For every digit of the caller the `.and()` function compares the corresponding digit of the argument. If both digits are 1 the `.and()` function returns 1, otherwise 0. An example of `45.and(5)` is shown below:

	0	0	1	0	1	1	1	0
and	0	0	0	0	0	1	0	1
	0	0	0	0	0	1	0	0

This `.and()` operation results in the binary number 00000100 which equals to 4 in decimal representation.

AirGuard first applies `.and(0x30)` onto every status byte, as seen in line 25 (Figure 22). 0x30 is 30 in hexadecimal representation which equals to 00110000 in binary representation. Therefore, the `.and(0x30)` operation acts as a filter, which only allows the bits at index 4 and 5 to go through. An example is shown below. The result is 48 in decimal representation.

	1	1	1	1	1	1	1	1
and	0	0	1	1	0	0	0	0
	0	0	1	1	0	0	0	0

After the `.and(0x30)`, the `shr` bitshift is executed. This operation also works on the binary representation of a number. It shifts all bits to the right by the specified number. Continuing with the result from above, 48 in binary representation is 00110000. Applying a shift to the right leads to 0011000. Further examples are shown in Table 6.

AirGuard always applies `shr 4` on every input, as seen in line 25 (Figure 22). Since before the `.and(0x30)` operation has been applied, those bits which get lost due to `shr 4` are always 0. Also, the two preceding bits are always 0. Hence, those two operations combined basically reduce the input from 8 bits to 2 bits, where only the information of those remaining two bits is stored. This leads to four possible cases, eiter 00, 01, 10 or 11. Those four possible results eventually represent Apple's device types as shown in Table 4, from which an AirTag can be identified.

Table 6: Examples of *shr* which shifts bits to the right.

00110000 shr 1	=	0011000
00110000 shr 2	=	001100
00110000 shr 3	=	00110
00110000 shr 4	=	0011

4.5.2 Chipolo ONE Spot

The identification of the Chipolo ONE Spot works in the same way as the one for the AirTag, as it is part of the Find My Network. Therefore, it is identified as an AirTag, as explained in subsection 4.5.1. The only difference is that the Chipolo ONE spot has a different device type according to Table 4. If the bits are equal to 0b10 the device is identified as a Chipolo ONE Spot.

4.5.3 Galaxy SmartTag+

The identification of a Galaxy SmartTag+ is fairly simple. It adds its name to the BLE advertisement. An example of its advertisement is given in the appendix (see Figure 41). It shows, that the `scanRecord` of the Galaxy SmartTag+ contains the local name of the device, which is "Smart Tag". If this name is found in the `scanRecord` of a BLE signal, the device is identified as a Galaxy SmartTag+.

4.5.4 Tile

A Tile tracker can be identified by a UUID. A UUID is a universally unique identifier which can be created independently [29]. The Tile tracker includes its offline finding UUID in the BLE advertisement [50]. An example is listed in the appendix (see Figure 44). The UUID equals to 0000FEED-0000-1000-8000-00805F9B34FB. If this UUUID is found in the `scanRecord` using the function `getServiceUUIDs()`, the tracker is identified as a Tile tracker.

As previously mentioned, the Tile app allows one to pair different BLE devices such as HP laptops, fitness trackers from Fitbit, headphones from Skullcandy, and many more. Unfortunately, no such device was available for this thesis. Hence, it could not be studied, if those devices also include the same UUID as the Tile tracker itself. Furthermore, Tile has been contacted, to get more information on this. Unfortunately, there was no reply.

Chapter 5

Results and Evaluation

For this work, several experiments were conducted. Those can be distinguished into two main categories. The first category is about the BLE advertisements of the tracking devices. Section 5.1 explains those experiments in more detail. The second category includes experiments to test the tracking algorithm. This is described in the section 5.2.

Since especially the experiments with respect to the tracking algorithm have been held in an iterative manner, the results are presented and discussed right away. This simplifies the thought processes which have taken place between two consecutive experiments. Finally, the evaluation of the holistic tracker identification is described in Section 5.3.

5.1 Advertisements of Tracking Devices

This section describes the conducted experiments regarding the informational content of the tracker's BLE advertisements. It includes experiments regarding the tracker's MAC addresses and how they randomize it in different scenarios, as described in Section 5.1.1. Furthermore, in Section 5.1.2 the RSSI (Received Signal Strength Indicator) value of an advertisement is measured and evaluated within different scenarios.

5.1.1 Long Time Unpaired vs. Alternating Pairing State

It is assumed that an adversary acts in two different scenarios to track a victim's location. In the first scenario, the adversary is at a remote location, thus not within the connection range of the BLE tracker in use. This results in the tracker remaining in the unpaired state for a long period of time. The second scenario involves the adversary actively following the victim. This way, depending on the distance between the adversary and the victim, the BLE tracker might be within or not within the connection range. So the BLE tracker alternates between the paired and unpaired state, perhaps multiple times. For both scenarios, the measurements have been restricted to one hour. This interval of one hour is derived from the fact that on average people in Switzerland commute 29

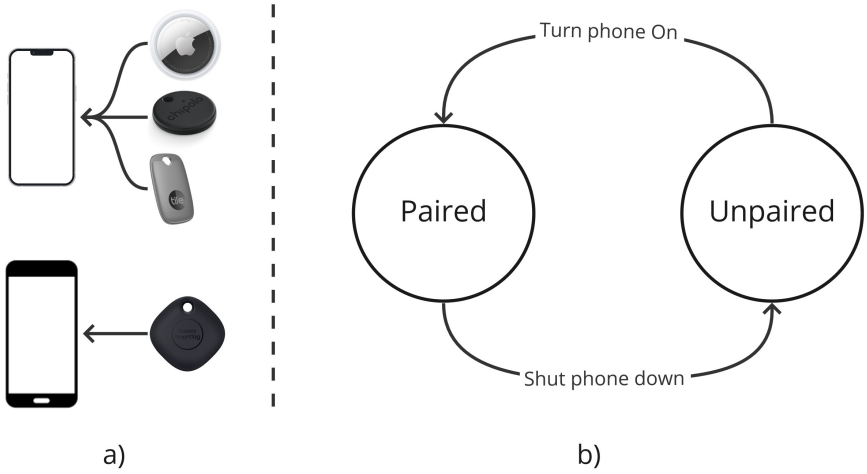

Figure 23: Experimental setup. a) shows the pairing of trackers with an iPhone (above) or Samsung Galaxy (below). b) shows how the trackers transition between paired and unpaired states.

minutes between their homes and workplaces [51]. Furthermore, in Switzerland on average people work 1'495 hours per year [52] which is a little less than 29 hours per week. This makes up almost 70% of a full-time job working 42 hours per week. Hence, on average Switzerland's population works 70% of a working week and has to commute 30 minutes one way, this commuting scenario affects many of Switzerland's residents and is therefore put into focus for those experiments, respectively supports the decision to measure the BLE advertisements for one hour.

Overall, the unpaired state of a tracker is regarded as the state where a BLE tracker is lost. In case of Apple's AirTag, its advertisements can then be picked up by anyone. In case an iPhone receives this advertisement, it creates a location report for the owner, as described in section 3.2.1. Those advertisements from all trackers used in this work are under investigation for this experiment. Especially, the MAC address of those advertisements are of core interest in those experiments.

Before tracking devices can transition into their lost state, they need to be paired with an owner. The AirTag, Chipolo ONE Spot, and Tile trackers have been paired with an iPhone SE running on iOS version 15.6.1. The Samsung Galaxy SmartTag+ on the other hand needed to be paired with a Samsung Galaxy. The tracker has been paired with a Samsung Galaxy A51 which runs on Android Version 11. To transition the trackers into the lost state, the smartphones had been shut down. To transition back to the paired state the smartphone had been turned on again. The experimental setup is depicted in Figure 23.

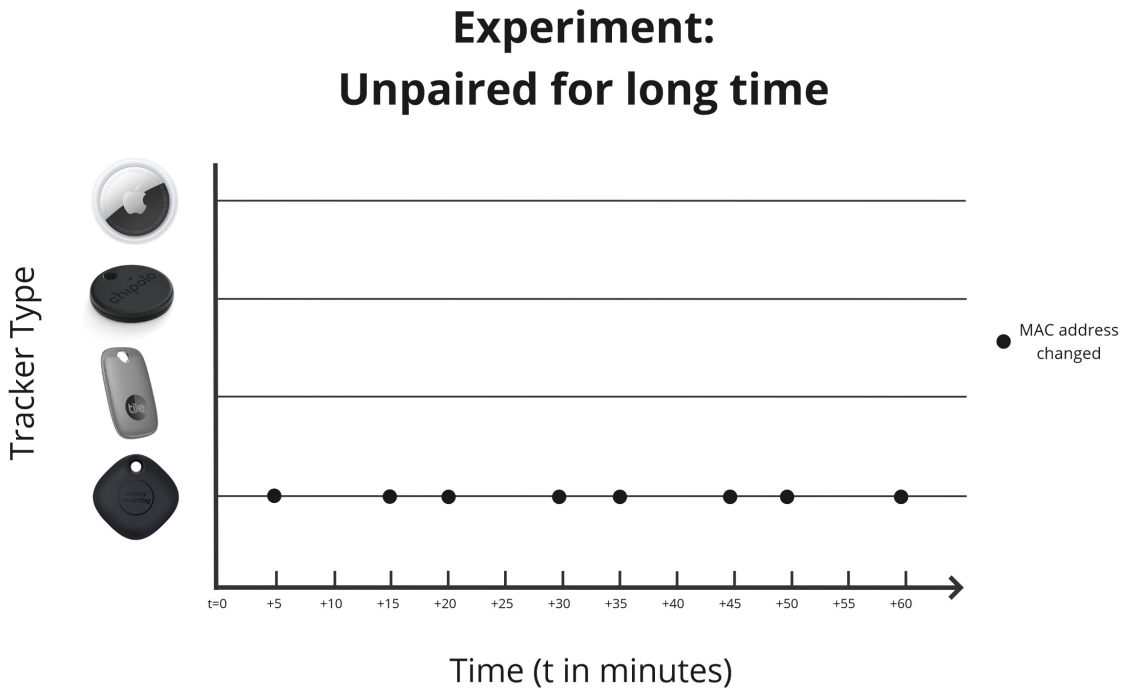

Figure 24: Results of recorded MAC addresses per Tracker while being unpaired from the owner. Only the Samsung Galaxy SmartTag+ applies MAC randomization while unpaired.

Long Time Unpaired

The idea behind the Long Time Unpaired experiment is that a tracker is not within connection range to the owner's phone. Therefore, the owner, respectively the adversary locates a victim from a remote location. To simulate the scenario in this experiment, the smartphones had been turned off for the full hour. Within this hour, the BLE advertisements had been scanned for all trackers within an interval of 5 minutes. Only the MAC address of the trackers had been recorded in this experiment. All the other attributes were left out. This experiment answered how the trackers randomize their MAC address when they are unpaired for a longer period of time. Figure 24 depicts those results.

The main insight of this experiment is, that only the Samsung Galaxy SmartTag+ randomizes its MAC address while being in the unpaired state. It does so in an interval between 5 to 10 minutes. The AirTag, Chipolo ONE Spot and the Tile tracker have advertised always the same MAC address during this experiment. Thus, it is further assumed within this work that they do not randomize their MAC address while being in the unpaired state.

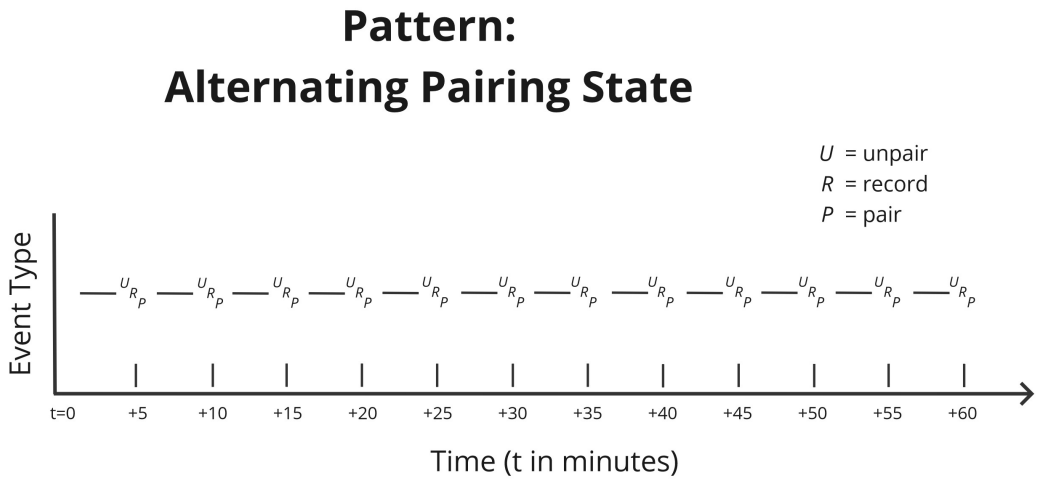

Figure 25: Pattern of alternating pairing state experiment.

Alternating Pairing State

The experiment where a tracker alternates between the paired and unpaired state represents a scenario, where the adversary is following the victim closely. For example, the victim is in a city and the adversary is maybe a block or two behind. In such a scenario, the adversary might be close enough to the victim such that the tracker is in connection range. This is possible considering the accuracy of location reports and the estimated range of BLE signals in urban and outdoor areas.

In case of Apple's AirTag an accuracy of 30 m has been deduced in urban areas while walking [19]. As the Chipolo ONE Spot is included in Apple's Find My network, the same accuracy for it is assumed. For the Galaxy SmartTag+ and Tile, no indication of the accuracy has been found. However, finding a SmartTag+ and a Tile works with an offline finding network as well, thus it is assumed that the accuracy is also around 30 meters. In a nutshell, the victim's actual position lies within a radius of 30 meters from the reported location. This means, in the worst case where the error of the location report is 30 meters, the victim might be 30 meters closer to the adversary as expected.

The estimated range of BLE advertisements broadcasted over PHY LE 1 in industrial and outdoor areas lies between 30 to 75 meters [53]. Now, assuming the worst signal strength which only ranges 30 meters, in combination with an accuracy error of the location report of 30 meters, and the adversary following the victim with a distance of 50 meters. This leads to two cases. First, it might be possible that the victim and adversary are only 20 meters apart due to the accuracy error. In this case, the adversary is definitely in connection range with its tracker. Secondly, it might be possible that the victim and adversary are 80 meters apart due to the accuracy error. In this case, the adversary is not in connection range with the victim. Therefore, during a stalking attack switching

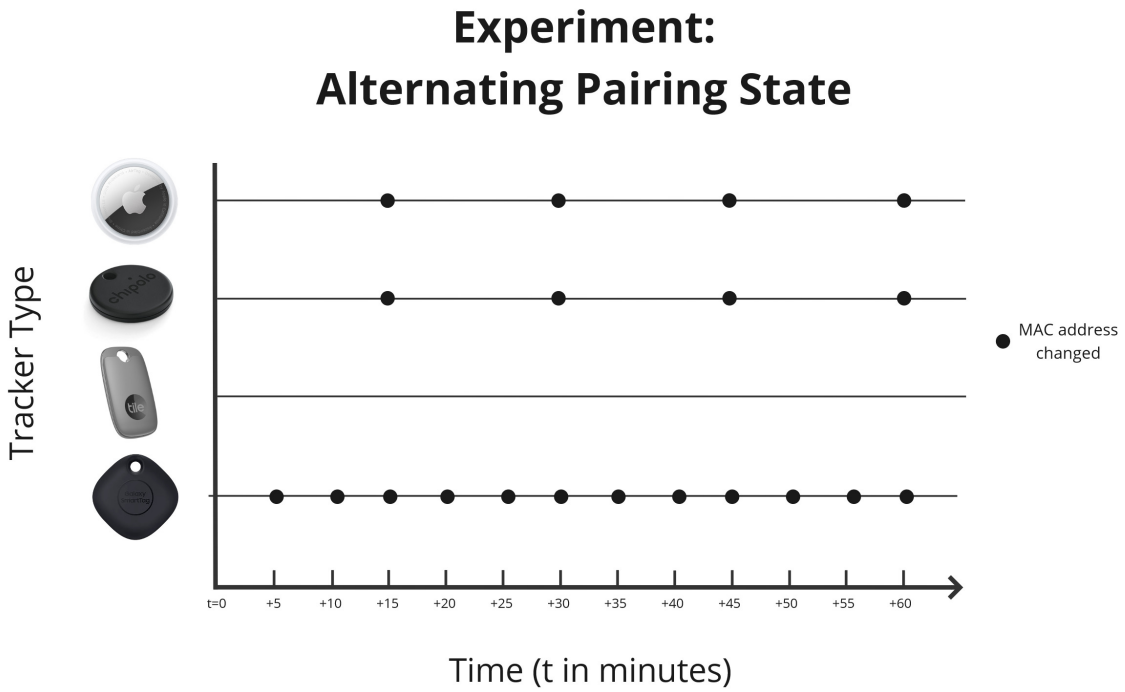

Figure 26: Results of recorded MAC addresses per tracker being unpaired while alternating states as stated in Figure 25. Only the Tile does not apply MAC randomization.

between cases one and two is possible. Hence, the possibility, that the tracker transitions between the paired and unpaired state is likely and therefore considered in this experiment. It needs to be noted, that the stalking distance of 50 meters has been chosen randomly. There is no evidence of how much distance an adversary should keep from a victim while stalking.

The trackers transition from the paired to the unpaired state by turning down the smartphones. While the trackers are in the unpaired state, their MAC addresses are recorded. After a measurement, the trackers are paired with the smartphones again. This pattern has been repeated every 5 minutes, as shown in Figure 25. The results of the experiment showed that only the Tile tracker does not randomize its MAC address, as shown in Figure 26. In fact, it never has randomized its MAC address at all. In both experiments, which lie one week apart, the Tile tracker has always broadcasted the same MAC address as it can be derived from the measurements in the Appendix B.3. The AirTag and the Chipolo ONE Spot on the other hand have started to randomize their MAC addresses compared to the Long Time Unpaired experiment. This leads to the assumption that the MAC randomization is induced by the iPhone. More precisely, the iPhone randomizes the tracker's MAC address every 15 minutes. Looking at the Galaxy SmartTag+ it is interesting to see that the randomization of its MAC address has increased. In this experiment, it changed its MAC address every 5 minutes, whereas in the long time unpaired experiment the randomization interval was between 5 to 10 minutes.

Table 7: Received Signal Strength Indicator (RSSI) values measured in dBm for different Scenarios with Android's BLE API.

		Scenarios (approximate distance)						
		On the phone (0 m)	In the jacket (0.3 m)	In the bag (0.5 m)	Edge of table (1.5 m)	Corner of room (4 m)	Other room (10m)	Size of range |max|-|min|
Trackers	AirTag	-40.7	-59.3	-65.8	-76.1	-77.7	-89.9	49.2
	Chipolo ONE Spot	-45.9	-62.2	-61	-72	-82	-93.9	48
	Galaxy SmartTag+	-40	-49.3	-51.5	-58.3	-73	-89.2	49.2
	Tile	-31.1	-50.2	-54	-60.2	-64.6	-86.6	55.5
	Mean	-39.425	-55.25	-58.075	-66.65	-74.325	-89.9	50.475

5.1.2 RSSI

The experiments regarding the Received Signal Strength Indicator (RSSI) value have been done out of curiosity. One vague idea was, that the RSSI value might help to keep BLE trackers apart, respectively helps to identify them. Another aspect was, to assess the RSSI's reliability of the trackers used within this work. In short, those experiments were a shot into the blue.

Within those experiments, the trackers' RSSI values have been measured with Android's BLE API. The trackers have been placed away from the scanning smartphone at several distinct distances. The owning smartphones had been turned off, such that the trackers transition into their unpaired state and broadcast their BLE advertisements.

In total six scenarios, the RSSI values have been recorded, as depicted in Table 7. The scenarios "In the jacket" and "In the backpack" are meant to represent a real-life scenario where the adversary has slipped a tracker into the victim's jacket or backpack. The scenarios "On the phone", "Edge of table", "Corner of room", and "Other room" do not represent a real-life situation which can be associated with a tracking scenario. Those scenarios have only been included, to assess the reliability of the RSSI with more data points.

The main goal was to figure out how the RSSI values behave for distinct, respectively increasing distances. Therefore, for every tracker and every distinct distance ten measurements of the RSSI value have been recorded. From those results, the average of every tracker in each scenario was computed. Table 7 shows this data. It furthermore includes the mean among all trackers. Besides that, for every row, the range of the RSSI values with respect to all distinct distances is listed.

Figure 27 depicts the RSSI values and their mean as graphs. The mean is regarded as a trend line. It shows that the larger the distance, the lower the RSSI value becomes. It can also be noted that there exists a steeper decline from the scenario "On the phone" (0 m) to the scenario "In the bag" (0.5 m) compared to the decline between the remaining scenarios (0.5 - 10 m). Generally speaking, the mean indicates that the signal strength of a BLE advertisement loses very much within the first 50 cm and declines less with a larger distance. Finally, each tracker never crosses the trend line. Either all its values are above or below the trend line. Hence, the RSSI values of an individual tracker steadily decline, as the trend line does.

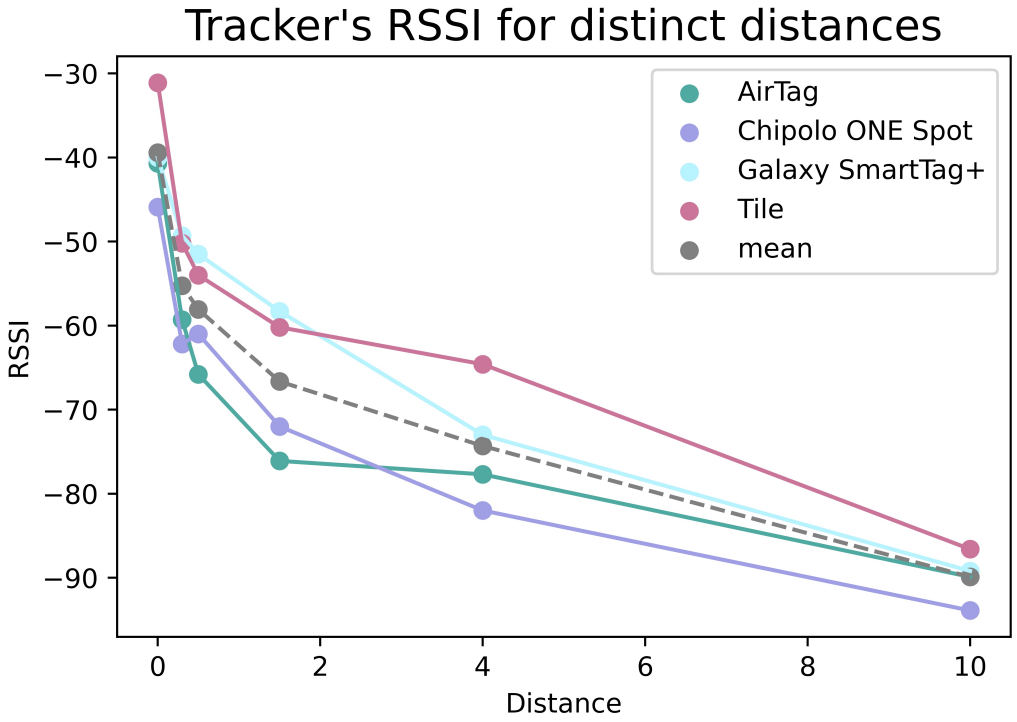

Figure 27: Points representing measured averaged RSSI values at a certain distance for every tracker. The line connects the points of a tracker to show the trend of the data.

Looking at the size of the ranges of the individual trackers, as shown in Table 7, it describes the difference between the lowest and highest RSSI value measured for each tracker. The Chipolo ONE Spot shows the smallest range in RSSI with a value of 48. The AirTag and the Galaxy SmartTag+ show the same range of 49.2. The Tile tracker shows the biggest range in RSSI with a value of 55.5. The range of the Tile tracker is nearly 10% larger compared to the mean. The other trackers deviate in their range from the mean by approximately 2.5%. The range of the Tile tracker is therefore approximately 4 times higher with respect to the mean compared to the ranges of the remaining trackers. This does not speak for reliable RSSI values among all trackers in a general manner.

Furthermore, an effect is noticeable, which will be called the blow-up-convergence effect. It describes, that the range between the trackers is rather small in the first two scenarios. Afterwards, the range expands or blows up. For the scenario ”Edge of table” and ”Corner of room” all RSSI values lie within a range of 17.8, respectively 17.4. At a distance of 10 m, the RSSI values of all trackers converge to a value between -93.9 and -86.2, respectively a range of 7.7.

Looking at the individual graphs, a second effect can be observed, which will be called the crossing-effect. It describes, that the RSSI graph of the Tile tracker crosses the RSSI graph of the Galaxy SmartTag+. This means, in the scenario ”Edge of table” the RSSI value of the Tile tracker is below the one of the Galaxy SmartTag+. In the following scenario ”Corner of room”, the RSSI value of the Tile tracker is above the one from the Galaxy SmartTag+. This is also true for the AirTag which crosses the values of the Chipolo ONE Spot between the scenarios ”Edge of table” and ”Corner of room”. Thus,

the behaviour of the Tile tracker and the AirTag is for those scenarios contrary to the observed trend line. Both trackers show a lower decline in their RSSI values.

Moreover, the Chipolo ONE Spot also shows behaviour which deviates from the trend line. Between the scenario "In the jacket" and "In the bag" its RSSI value increases from -62.2 to -61.

Finally, all observations are based on an averaged value out of ten measurements for every tracker. Since even those averaged results of the individual trackers show strange behaviour, it is expected that looking at a single experiment, more strange behaviour would be revealed. This leads to the assumption that the RSSI signal on an individual level is not reliable or stable.

To summarize, on one hand, the experiments regarding the RSSI values show promising results that in general, the RSSI value decreases for every tracker with a larger distance. On the other hand, the blow-up-convergence effect, the crossing-effect, or the sizes of the ranges show, that the RSSI value is rather unreliable or unstable. Those observations are based on averaged results and it is further assumed, that on an individual level, even more, strange behaviour of the RSSI value would be revealed. Hence, with the information from the RSSI values, no pattern can be derived, which helps to distinguish trackers from each other or to identify a tracker based on its RSSI values.

5.2 Tracking Algorithm

The Android app which has been developed for this work allows the user to individually set his or her tracking preferences. Those preferences consist of the minimal time and distance a tracker needs to follow the user, as well as the minimal number of times a tracker needs to be scanned. Once all those values are exceeded the user gets notified. Hence, the app developed for this work allows the investigation of those three preferences. It can be examined which selection for those values brings the most reliable results, such that the tracking algorithm notifies the user as early as possible about malicious trackers. The following subsections describe the route taken to examine the tracking algorithm and the procedure while walking this route. Afterwards, the individual experiments are described and evaluated, which vary in their selection of the tracking preferences. The results are always presented and discussed because the follow-up experiment is based on the previous one.

5.2.1 Experimental Route

The chosen route depicts three possible scenarios in which a victim might be tracked. As described in Figure 28, the starting point of the experimental route is surrounded by grocery stores, a local bar as well as a public place, which is frequently visited. The destination is regarded as the victim's home. It is assumed, that the starting point is of interest for an adversary to track a victim. An adversary sees or meets a victim at the grocery store, bar or public place, and wants to follow the victim back to his or

Figure 28: Shows the route chosen to test the tracking algorithm. According to Google Maps, the route is 500 m long and it takes about 5 min to walk from start to finish. The black play icon represents the starting point. The black flag icon represents the destination. The route can be subdivided into three parts as indicated by the colors red, blue and green. Within the red part, there is a Coop and Migros, a bar and a public place. The blue part is a path along a train station, which is more open and therefore has fewer objects blocking a BLE signal. The green part fully represents a residential area with many apartments and a recreational park.

her home. To recall, the starting point is a stationary location of the victim, where he or she does not cover a large distance, respectively 50 meters, as declared in the `LocationTrackingService` described in Section 4.3.3.

5.2.2 Experimental Procedure

For each walk of the experimental route, the user was equipped with the app and enabled the tracking protection feature before heading home from the starting point. The user, respectively the victim was carrying all four trackers used within this work in the left pocket of his or her jacket. The Android smartphone, a realme 9 Pro 5G running on Android Version 12, had been carried in the right pocket of the user's jacket. The app's tracking protection remained turned on until the `LocationTrackingService`, as described in Section 4.3.3, evaluated that the user is not moving anymore. This always resulted in the tracking app to stop scanning for BLE devices a few minutes after the user has returned home. Furthermore, the user was always walking alone, respectively without the company of another person. Lastly, the app's database was empty, meaning that no BLE devices or malicious trackers, as described in Figure 17, from previous runs were stored in the database.

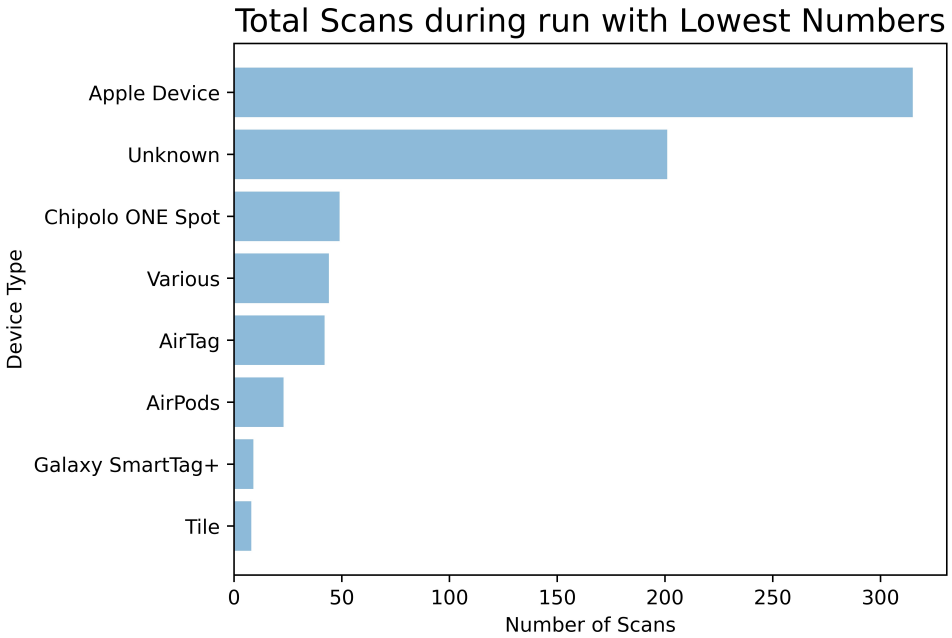

Figure 29: This bar chart plots the number of times a device type got scanned during the run with the lowest number of scanned devices. In total, the run recorded 698 BLE scans which map to 432 distinct BLE devices according to their MAC address. The device type Various groups devices which included their name in the BLE Advertisement.

5.2.3 BLE Devices in the Wild

Before jumping into the experiments regarding the tracking algorithm, an idea is given about what kind of BLE devices during a run of the experimental route have been scanned. In total, 15 runs have been performed. These runs were conducted at random times of the day. This implies that the BLE traffic varied during those experiments, respectively the number of scanned devices varies. The run with the lowest number of scanned devices, as well as the run with the highest number of scanned devices, are depicted in Figure 29, respectively Figure 30. For each of the mentioned runs, the types of devices and how many times such a device appeared are shown.

In both cases, Apple devices and devices of Unknown type are the most scanned ones. The Tile tracker and the Galaxy SmartTag+ show in both cases the lowest number of occurrences. AirTags and Chipolo ONE Spots exceed those occurrences by far. Hence, it is assumed that during those runs, foreign AirTags and Chipolo ONE Spots have been scanned, respectively trackers of those types which have not been carried by the user. For the Galaxy SmartTag+ and the Tile tracker, it seems that only the carried-along trackers have been out in the wild during those runs.

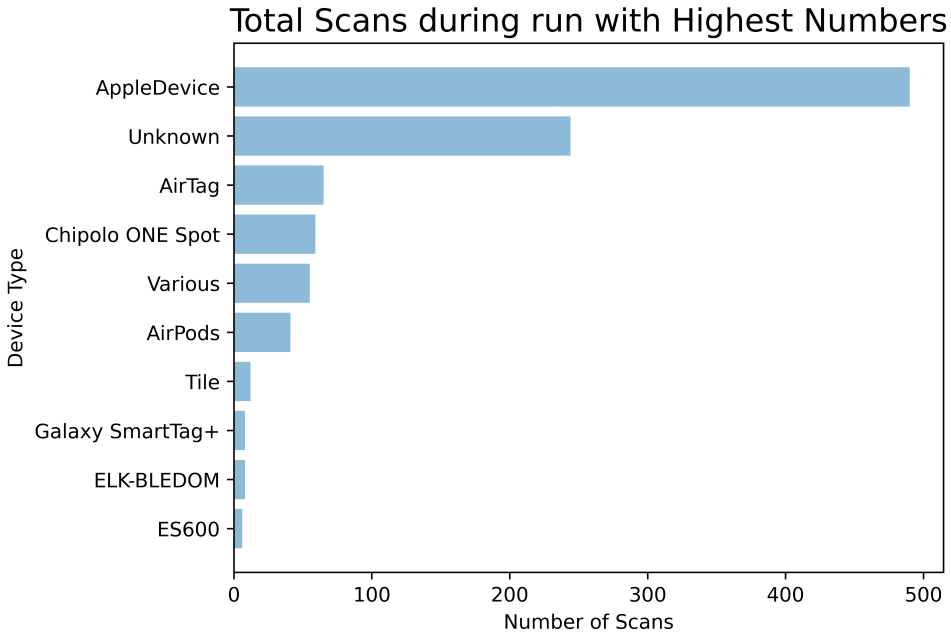

Figure 30: This bar chart plots the number of times a device type got scanned during the run with the highest number of scanned devices. In total, the run recorded 988 BLE scans which are mapped to 644 distinct BLE devices according to their MAC address. The device type Various groups devices which included their name in the BLE Advertisement.

5.2.4 Testing of Tracking Preferences

Since three parameters are in place to decide if a BLE device represents a malicious tracker, the analysis of which value to choose for every parameter happens individually and in an iterative manner. This means, in the first experiments only the parameter distance varied, where the number of occurrences, as well as the time preference, remained the same.

As described in Section 4.3.3 about the services of the tracking algorithm, the parameter regarding the distance can decide about a tracker being malicious by having only two scans of a tracker at hand. The same goes for the time parameter. Contrary, the parameter occurrences depends on more than two BLE scans if its value is increased by the user. This implies that the higher the number of occurrences is set, the more time needs to pass such that enough scans occur. Only after this, the tracking algorithm can classify a device as a malicious or non-malicious tracker. Therefore, the occurrences parameter has not been chosen to be the first parameter to evaluate in the experiments.

A decision had to be made to determine whether to start testing the tracking algorithm with the distance or time parameter first. One disadvantage of the time parameter is, that the selected time must pass before a user gets notified about a malicious tracker. It does not matter how fast or slow a user is moving. The distance parameter on the other hand does not have this artificial delay of a notification. The app allows setting a minimal distance of 50 meters, a distance which is probably covered below one minute. Hence, the parameter distance had been chosen to be the first to analyse. It is assumed, that it

Table 8: Results of the experiments to test the tracking algorithm with an increasing distance value. The time parameter and the occurrences parameter were left at their minimal values, which is 1 minute and 2 occurrences.

		Results						
		True Positives	False Positives	False Negatives	True Negatives	Precision	Recall	Total scanned devices
Distance	50 m	4	23	0	470	14.8%	100%	497
	100 m	4	9	0	497	30.7%	100%	510
	150 m	3	3	1	470	50%	75%	477
	200 m	4	0	0	471	100%	100%	475
	250 m	4	0	0	520	100%	100%	524

might have the biggest impact, such that a user can be notified about malicious trackers as early as possible.

Finally, for every experiment, the true positives, false positives, true negatives and false negatives have been determined. The true positives represent the four trackers carried along in the left pocket of the jacket that need to be identified by the tracking algorithm. The false positives represent scans of BLE devices which are picked up during a run and which are classified as malicious trackers. However, those false positives are in reality not used as a BLE device to track the user. A false negative is one of the four BLE trackers used in this work, which has not been classified as a malicious tracker. Finally, the true negatives are all the BLE signals which have been captured by the app during a run but were correctly classified as a non-malicious tracker. From those results, the precision and recall can be calculated, which are computed for every experiment as well.

Distance

By design, the app allows one to choose a minimal distance of 50 meters. The step size is 50 meters. The maximal value is 1000 meters. Therefore, the first experiment has been conducted with the lowest value for the distance parameter, respectively 50 meters. Afterwards, for every consecutive run of the experimental route, the value for the distance parameter has been increased according to the step size. The values for the parameters time and occurrences were left at their lowest possible value which is 1 min for the time parameter and 2 occurrences. In total 5 runs have been completed. The results for the varying distance parameter, the true positives, false positives, false negatives, true negatives, precision, recall, and the total number of scanned devices in each run are depicted in Table 8.

The experiments only on the distance parameter showed that using a value of 200 meters results in the correct classification of malicious and non-malicious trackers. All carried-along trackers were identified and no other, respectively no false positives occurred. An extra run with an increased distance value equal to 250 meters showed also correct results. It is assumed that increasing the distance parameter, even more, would still lead to correct results. However, this leads to classification problems with respect to a different route. A

Table 9: Results of all three runs with distance parameter set to 200 meters. The time and occurrences parameters are set to their minimal values which are 1 minute and 2 occurrences.

	Results						
	True Positives	False Positives	False Negatives	True Negatives	Precision	Recall	Total scanned devices
First	4	0	0	471	100%	100%	475
Second	4	0	0	524	100%	100%	528
Third	4	1	0	456	80%	100%	461

simple example would be a shorter route of 300 meters and a distance parameter set to 500 meters. In such a scenario, the malicious trackers would not be detected. Thus, since it is assumed that larger values for the distance parameter still provide correct classification results, but lead to wrong results depending on the length of the experimental route, choosing the smallest possible value for the distance parameter fits best. Derived from the experiments taken so far, this leads to a selection of 200 meters for the distance parameter.

Taking only a distance parameter of 200 meters already shows promising results. To further decrease the distance parameter to 150 meters, the time and occurrences parameter will be taken into consideration. Therefore, in the corresponding follow-up experiments, the distance parameter is tried to be decreased to 150 meters, but not lower. This way, the tracking algorithm can be configured with the parameters which take effect as early as possible, respectively notify the user about malicious trackers as early as possible.

Despite the tracking algorithm classifying correctly with the selection of the distance parameter of 200 meters, it needs to be pointed out that the run with the parameter set to 150 meters showed a false negative, respectively did not classify the Chipolo ONE Spot as a malicious tracker in this very run. This can be regarded as bad luck due to the scanning frequency. The tracker was not detected early enough, thus its first and last occurrences do not show a difference in the distance above 150 meters. Nonetheless, this raised the assumption that with the distance parameter set to 200 meters, the classification might be correct by chance. Therefore, another two runs with the distance parameter set to 200 meters were executed. The results of all those three runs are depicted in Table 9. The decision not to repeat the runs with the distance parameter set to 150 meters is based on the fact that in this run still 3 false positives occurred. Thus, the sum of false results within the run of 150 meters is 4, whereas the sum of false results within the run of 200 meters is 0. Therefore, the experiments were repeated with the distance parameter of 200 meters and not 150 meters.

As seen in Table 9, all three runs show a recall of 100 %. Therefore, the assumption, that the false negative from the run with the distance parameter set to 150 meters was bad luck due to the scanning frequency hardens. Nevertheless, the last run shows a lower precision of 80 %, which means a false positive occurred. It was an Apple device which was classified wrongly as a malicious tracker. This device had two occurrences during the experiment. It got first scanned at lat: 47.393039, lon: 8.5291302 and afterwards at lat:

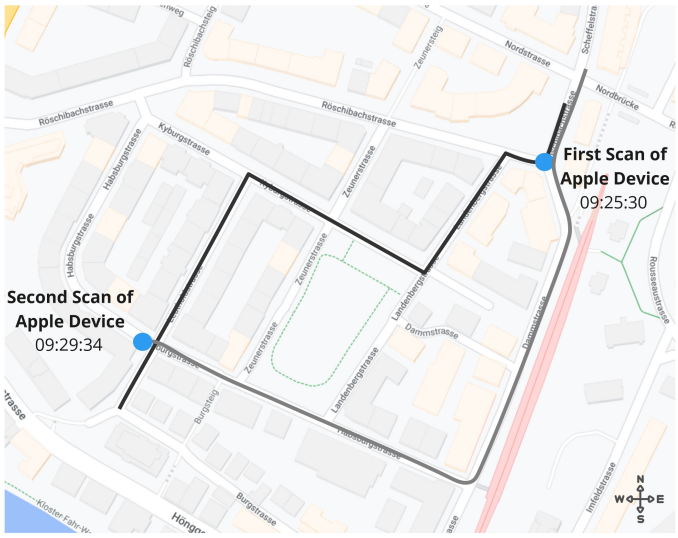

Figure 31: The third run, as seen in Table 9, testing the tracking algorithm with respect to the distance parameter showed a false positive. It was an Apple device which crossed the experimental route. The device got scanned exactly twice during the experiment. The locations of both scans are indicated with the blue circle and the according timestamp. A potential route of this Apple device taken is drawn with a black line. The route taken in the experiment is drawn with a grey line. According to Google Maps the route of the Apple device is 350 meters long and walking this route takes approximately 5 minutes. Hence, the owner of the Apple device probably has taken this route at the same time the experimental route was taken and finally, the crossing happened at the destination of the victim.

47.3921147, lon: 8.5258758. Checking those coordinates using Google Maps, the device appeared at the beginning of the experimental route and at the destination, respectively the home of the victim. Figure 31 shows where the Apple device got scanned, and a potential route the Apple device could have taken. The timestamp of the first scan is at 1669278330905 milliseconds which is the 24th of November 2022 at 09:25:30. The second scan was recorded at 1669278574751 milliseconds, which is the 24th of November 2022 at 09:29:34. Therefore, the route the Apple device has taken is roughly 4 minutes long. This is plausible, Google Maps states that this route is approximately 5 minutes long. Hence, the false positive of the Apple device perhaps happened, because the person carrying the Apple device got crossed at the beginning of the experimental route and at the end of the experimental route. Therefore, for this experiment and its set parameters, it can be noted, that false positives can occur, when people cross the user on his or her way, as this is shown with the aforementioned example of the Apple device.

Investigating further on the occurrences of those three experiments, the true positives, respectively the actual BLE trackers showed 6 (Tile), 8 (Galaxy SmartTag+), 11 (Chipolo ONE Spot), and 12 (AirTag) occurrences. In the second run the trackers showed 5 (Tile), 6 (Galaxy SmartTag+), 8 (AirTag), and 9 (Chipolo ONE Spot) occurrences. Finally, in the last run the devices got scanned 4 (Galaxy SmartTag+), 7 (AirTag), 8 (Tile), and 9 (Chipolo ONE Spot) times. Considering all three runs, the range in occurrences for the

AirTag is 5, for the Chipolo ONE Spot 2, for the Galaxy SmartTag+ 4, and for the Tile 3. The AirTag shows the largest range, but the Galaxy SmartTag+ and the Tile Tracker show the lowest number of occurrences which is 4 and 5 respectively.

Since it is known from the experiments regarding the tracking devices in Section 5.1 that the Galaxy SmartTag+ randomizes its MAC address, it is not surprising that it showed the lowest number of occurrences. It might be possible that the Galaxy SmartTag+ changed its MAC address during the experiment. For the Tile tracker on the other hand it might be explained that it has a larger advertising interval, which means it sends out its BLE advertisement less frequently. Therefore, it shows a lower number of occurrences. This would need to be tested in a further experiment to confirm the assumption. However, those numbers on the occurrences of all the true positives lead to the assumption that the parameter for the occurrences can be raised up to 4 occurrences in order to get rid of the false positives, while still keeping the true positives. This number has therefore been chosen as a base for the experiments regarding the occurrences which are described in the following subsection.

Also, it has been decided not to undertake any experiments regarding the advertising interval of the Tile tracker, since the insights gained from those experiments would only fit the Tile tracker and none of the other trackers. Thus, it would not improve the tracking algorithm in a general manner.

Occurrences

So far, setting the distance parameter to 200 meters shows good results in classifying signals into malicious and non-malicious trackers. With the help of the occurrences parameter, it will be tested if the distance parameter can be decreased to 150 meters. From the `BluetoothScanningService`, as described in Section 4.3.3, it is known that the app scans for trackers twice within a minute with a scan period of 12 seconds. The maximal scanning interval of BLE advertisements is 10.24 seconds [27]. Thus, it can be assumed that every tracker is at least scanned twice within a minute, respectively showing two occurrences. Raising the value for the number of occurrences therefore also delays the point in time when the user gets notified about a malicious tracker. Therefore, to still be able to inform the user as early as possible, it is beneficial for the user to keep the number of occurrences as low as possible.

From the previous experiments, it could be shown that the true positives always showed at least 4 occurrences. Therefore, two experiments will be executed. The first run raised the occurrences to 4, whereas in the second run, the occurrences are set to 5. Using those values for the occurrences parameter should allow the app to identify malicious trackers below 3 minutes. The distance value is set to 150 meters, and the value for the time parameter was left at its minimal value, which is 1 minute. The results are shown in Table 10. For a better overall comparison, it also includes the run from the previous experiments with 150 meters for the distance parameter, 2 occurrences, and the time parameter set to 1 minute.

The first run has not detected all trackers. The reason for this has been discussed already in the experiments regarding the distance parameter as described above. The second and

Table 10: Results of all three runs with different occurrences parameter. The distance and time parameters remained constant. The distance parameter is set to 150 meters and the time parameter is set to its minimal value, which is 1 minute.

		Results						
		True Positives	False Positives	False Negatives	True Negatives	Precision	Recall	Total scanned devices
Occurr.	$\# \geq 2$	3	3	1	470	50%	75%	477
	$\# \geq 4$	4	2	0	426	66%	100%	432
	$\# \geq 5$	4	1	0	614	80%	100%	614

the third run with the occurrences above or equal to 4, respectively 5 have a recall rate of 100 %. However, all runs show false positives. In short, this means that the parameter for the occurrences is not capable to support the distance parameter, respectively allow the distance parameter to be decreased from 200 meters to 150 meters. Nonetheless, the higher the number of occurrences is used, the fewer false positives are recorded, respectively the precision increases.

At this point, it has been decided to stop investigating on a higher value for the occurrences parameter for two reasons. First, increasing the occurrences to 6 or more also postpones the point in time when the user will be notified about malicious trackers. In the worst case, it takes three minutes, because a tracker might only get scanned twice within a minute due to the design of the `BluetoothScanningService`. Because using a distance parameter of 200 meters, occurrences of 2 and a value of 1 minute for the time parameter has already shown good results, it does not make sense to further increase the value for the occurrences. Nonetheless, the design of the `BluetoothScanningService` in the tracking algorithm seems to restrict the usage of higher values for the occurrences parameter. However, the scan interval and scan period of the `BluetoothScanningService` could be adapted. An idea is discussed in Section 6.2.

The second problem with a higher value for the occurrences parameter might be a decrease in the recall rate. As described within the three experiments with a distance value of 200 meters and the occurrences and time parameter set to its minimal values, the true positives, respectively the Galaxy SmartTag+ and Tile trackers showed 4, respectively 5 occurrences only. Due to this fact, increasing the occurrences parameter more, also increases the likelihood of false negatives, because those two trackers would not be recognized as malicious trackers anymore. From a victim's point of view, false negatives impose a greater threat than false positives, since the victim would not be notified about a malicious tracker in contrast to being falsely notified about a tracker which is not a malicious tracker. To conclude, for those two reasons, no more experiments using 150 meters for the distance parameter and an increasing value for the occurrences parameter have been executed.

Table 11: Results of the three runs with occurrences parameter set to 4. The distance parameter is set to 200 meters and time parameter remained at its minimal value of 1 minute. The reason for those runs is, to figure if the occurrences parameter properly supports the distance parameter of 200 meters, where it performed best. This should support the distance parameter of 200 meters to eliminate false positives.

	Results						
	True Positives	False Positives	False Negatives	True Negatives	Precision	Recall	Total scanned devices
First	4	0	0	461	100%	100%	465
Second	4	0	0	471	100%	100%	475
Third	4	0	0	567	100%	100%	571

Despite the fact, that the occurrences parameter did not allow to decrease the distance parameter to 150 meters, it is still unanswered, if increasing the occurrences to 4 properly supports the distance parameter of 200 meters. As described within the experiments with respect to the distances parameter with a value of 200 meters, there was one run, where a false positive occurred. This false positive however only had two occurrences, whereas all true positives were scanned 4 or more times. Therefore, two additional runs of the experimental route have been taken using 200 meters for the distance parameter, 4 for the occurrences and the time parameters with its minimal possible value of 1 minute. The results are shown in Table 11.

All three runs show a precision and recall rate of 100 %. Therefore, the parameter occurrences with a value of 4 strengthens the performance of the distance parameter set to 200 meters. This can be assumed, since all three runs classified the malicious and non-malicious trackers correctly, compared to the experiments where the value of the occurrences was set to 2, as seen in Table 9. There, one false positive occurred, which is now eliminated.

Furthermore, the number of scanned devices in the current runs ranges from 465 to 571. In the earlier experiments, the range of scanned devices is between 461 to 528. Hence, the number of scanned devices is slightly higher for the current runs compared to the earlier ones. This also indicates, that the occurrences parameter supports the distance parameter and makes the results a little more robust, even when more devices are scanned during a run.

Finally, looking at the occurrences in the current runs, the AirTag was scanned 8, 7, and 8 times, the Chipolo ONE Spot 8, 10, and 9 times, the Galaxy SmartTag+ 7, 6, and 7 times, and the Tile was recorded 4, 8, and 8 times. The Tile tracker occurred within the first run only 4 times, which is the lowest number. Therefore, raising the occurrences would potentially produce false negatives which should be omitted.

To conclude, increasing the occurrences parameter to a value of 4 did not break the already good retained results with a distance parameter set to 200 meters. It even supports the distance parameter properly. Since also increasing the occurrences value more would lead to a potential delay of the point in time when the user gets notified about a malicious tracker, setting the value 4 for the occurrences parameter appears to be a good selection.

Table 12: Results of the three runs with the time parameter set to 2 minutes. The distance parameter is set to 150 meters and the occurrences remained at their minimal value of 2. The reason for those runs is to figure out if the time parameter is able to decrease the distance parameter to 150 meters.

	Results						
	True Positives	False Positives	False Negatives	True Negatives	Precision	Recall	Total scanned devices
First	4	3	0	637	57.1%	100%	644
Second	4	1	0	525	80%	100%	530
Third	4	3	0	437	57.1%	100%	444

Time

At this point, the experiments with respect to the distance and occurrences parameters, especially their combination, showed promising results in classifying malicious and non-malicious trackers. Hence, the question arises whether it is necessary to investigate on the time parameter. On one hand, due to the nature of the time parameter, it is assumed that the time parameter is rather a restricting parameter because it might delay the point in time a user gets notified about a malicious tracker. On the other hand, it makes sense to classify a BLE device as malicious or not, depending on the time it follows a user. This would clearly distinguish BLE signals from people passing by and BLE signals from trackers which actually follow the user. Lastly, the time parameter might help to decrease the distance parameter to 150 meters. Hence, the first argument advises against doing experiments with an increasing value for the time parameter. The other two arguments, however, suggest seeing if setting the time parameter to 2 minutes enables decreasing the distance parameter to 150 meters. Therefore, three runs are executed with a time parameter set to 2 minutes. The distance parameter is set to 150 meters and the occurrences are set to their minimum value of 2. The results are shown in table 12.

The results regarding the time parameter with a fixed distance and occurrences parameter show a recall rate of 100 % in all three runs. However, none of these runs show a precision rate of 100 %, thus in every run false positives occurred. In all runs, 7 false positives have occurred. This leads to the conclusion, that the time parameter does not help to decrease the distance parameter to 150 meters. Furthermore, the combination of the distance parameter set to 200 meters and the occurrences set to 4 provides good classification results. Thus, it seems that the time parameter can be neglected within this combination, as it would unnecessarily delay the point in time a user gets notified about a malicious tracker. For this reason, it has been decided to leave the time parameter at its minimal value of 1 minute.

Moreover, taking again another look at the occurrences of the Tile tracker, it has been recorded 4 times in the first run with respect to the occurrences parameter. The current runs with respect to the time parameter deliver some more data to analyse this number. The Tile tracker got scanned 10 times in the first, 8 times in the second, and 5 times in the third run. Hence, those numbers further harden the selection of the occurrences parameter with a value of 4, since the Tile tracker has never appeared less than 4 times.

Finally, it appears that there is an order of the parameters. The distance parameter on its own already provides good classification results. With the help of the occurrences parameter, those good results could be strengthened. The time parameter in the end seems to only delay the point in time where a user gets notified. Hence, the distance parameter can be regarded as the leading parameter, the occurrences can be seen as a supporting parameter, whereas the time parameter seems to be a restricting parameter. The distance parameter in this setting is set to 200 meters, the occurrences are set to 4 and the time parameter is left at its minimal value of 1 minute. This conclusion addresses (iii) as mentioned in Section 1.2.

5.3 Evaluation of Tracker Identification

Within this work, three ideas have been pursued to figure, if BLE advertisements can be distinguished into tracking versus non-tracking BLE devices, respectively how trackers can be identified as such. The first approach uses company identifiers. The second idea follows the work from Becker et al. [21], in which they found identifying tokens within the PDU's of an advertisement. Those approaches are done from the perspective of an Android smartphone. Finally, Android's BLE API is studied in more detail, to see if a holistic tracker identification is possible.

5.3.1 By Company Identifier

In Android, scanning for BLE devices returns so-called `ScanResults` that contain a `ScanRecord`. Examples are presented in Appendix A. One important property of it is the manufacturer-specific data. Using the function `getManufactuereSpecificData()` with the appropriate company identifier returns a `SparseArray<byte[]>`. Those company identifiers are uniquely defined by the Bluetooth SIG [26]. As an example, having an advertisement from an Apple device at hand and using the function call with Apple's company identifier `getManufactuereSpecificData(76)`, the manufacturer-specific data is returned as a byte array. Using the function with a different company identifier would return `null` in case of having an advertisement from an Apple device at hand.

The returned byte array is not interpretable for the human eye. Reverse Engineering is one approach, to break down this structure such that its information is accessible, as it has been done by Heinrich et al. [24]. However, reverse engineering BLE advertisements exceeds the focus of this work. Furthermore, as seen in the advertisement examples in Appendix A, not all BLE advertisements do include manufacturer-specific data by design. This is the case for the Tile tracker and the Galaxy SmartTag+. Thus, those devices can not be mapped to their manufacturer using the company identifier and no manufacturer-specific data could be used for tracker identification.

To conclude, manufacturer-specific data can not be interpreted right away in order to possibly figure out if an advertisement originates from a tracking device. Furthermore, the company identifier only tells, from which manufacturer the device is. It is not possible to learn if this device is a tracking or non-tracking BLE device.

5.3.2 By PDU

The address-carryover algorithm from Becker et al. [21] is based on so-called identifying tokens. For this, they used log files and decoded the BLE advertisements, respectively they observed the PDU Payloads as depicted in Figure 2. An identifying token might be for example a MAC address or a byte sequence in the PDU payload which does not change over time [21]. In this work, an Android app is used to scan and identify BLE devices, respectively to interpret BLE traffic. Therefore, the app relies on the API's to process BLE traffic.

As of writing, besides the known Android BLE API, other APIs, libraries, tools or a PDU parser which can be used in an Android application to investigate BLE advertisements are not known. Also, Android's BLE API does not allow to access raw PDU Payloads. The only track would be the function `getBytes()`, which returns the manufacturer-specific data. This however cannot be further used as explained above. In a nutshell, the address-carryover algorithm to track or identify BLE devices does not apply to this work as it is not possible to analyse PDU payloads in the same format as Becker et al. did.

5.3.3 Holistic

With the approaches described above, no tracker identification using Android's BLE API is possible. Nonetheless, the remaining properties the API provides, which have not been touched on in the two approaches above, are evaluated. From all those properties, two insights regarding the identification of BLE devices into tracking and non-tracking devices can be drawn.

Firstly, every BLE tracker advertises with an individual structure, as derived from the example advertisements in Appendix A. Apple's AirTag and the Chipolo ONE Spot, which has been designed as third-party accessory for Apple's Find My network, pack all their relevant advertisement information into the manufacturer-specific data. Using this data, those trackers can be identified, as described in Section 4.5.1. The Tile Tracker and the Galaxy SmartTag+ use service data to send information, which the AirTag and Chipolo ONE Spot do not use. Further, the Tile and the GalaxySmartTag+ advertise slightly different service UUID's. The Tile tracker is identified via its service data and the Galaxy SmartTag+ includes its name. To conclude, all trackers are identified in their individual way.

Secondly, for all the remaining properties, it is assumed that there is no pattern to distinguish BLE trackers from non-BLE trackers. Reasons for this assumption are, that many properties are undefined or do not show equal values. An undefined example is the function `getAdvertisingSID()`. None of the trackers has defined this value. Moreover, only the Tile tracker and the Galaxy SmarTag+ have values defined for the `getAdvertiseFlags()` function, but those properties are defined differently (Tile = 6, SmartTag+ = 4). Furthermore, only the Galaxy SmartTag+ returns a value for the `getType() = 2` function. According to Android's documentation, this value represents a BLE device. Other return types are classic, dual, or unknown. Those return values do not allow to distinguish tracking and non-tracking BLE devices.

Besides many further undefined properties or properties with sparse information, some remaining properties focus on physical attributes, such as the functions `getPrimaryPhy()`, `getTimeStampNanos()`, `getTxPower()`, and `getRssi()`, or they focus on version attributes like `isLegacy()`. Those attributes are assessed to be too soft to use as criteria for this identification task. Thus, Android's BLE API does not allow for a holistic identification of tracking vs non-tracking BLE devices.

Summarizing the insights regarding tracker identification, it has been assessed, that identifying trackers by company identifiers or PDU's as well as a holistic classification is not possible. The trackers used in this work can be identified down to their type as shown in Section 4.5. Hence, those 4 trackers can be identified as BLE trackers. Nonetheless, the identification of an AirTag, Chipolo ONE Spot, and Tile tracker is only possible due to heavy analysis of the corresponding BLE advertisements done by Heinrich et al.[24]. Within this work, the identification of a Galaxy SmartTag+ has been studied. Luckily, it did not require a lot of work, since it advertises its name. Furthermore, Tile has been contacted in order to retrieve information how to identify other accessories which they support within the Tile app, but no reply was received.

To conclude, at this point, identifying a BLE tracker, is only doable by studying the advertisements of every individual tracker, where the approach regarding the company identifier, respectively the manufacturer-specific data, as well as the idea using the PDU's is restricted from an Android perspective. Those insights address (i) as mentioned in Section 1.2.

Chapter 6

Discussion

This section discusses the results in order of the contributions of this work. Those contributions are listed in Section 1.2. The discussion ranges from security concerns of trackers, the app's scanning behaviour, additional false positives, MAC randomization, and how RSSI might be of further use for tracking protection.

6.1 Security Concerns of Trackers

A holistic identification of a tracker is not possible using Android's BLE API, as explained in Chapter 5.3. Each tracker needs to be analysed individually in order to identify it. This may include some heavy work as it has been done for the AirTag and Chipolo ONE Spot by Heinrich et al. [24]. Contrary, the Galaxy SmartTag+ and Tile tracker are rather identified fast in terms of workload. Hence, the AirTag and Chipolo ONE Spot are hard to identify, whereas the Galaxy SmartTag+ and the Tile tracker are easy to identify. In case more trackers will appear on the market, it might need heavy work in order to protect victims appropriately from stalking attacks using new BLE trackers. To avoid such heavy work, finding a way to holistically identify trackers still can be a good solution to solve this problem. This might be achieved by adapting the BLE protocols, respectively advertisements or it needs further studies to find a way which accomplishes this task. This would help victims to protect themselves from stalking attacks in an easy and future-oriented way.

In contrast, a holistic tracker identification might result in security issues regarding the intended usage of a tracker. The actual purpose is to find lost items, such as a key or perhaps a bag in which the owner stores his or her valuable items. Now, let's turn the tables, where an adversary tries to steal an item attached to a BLE tracker. A holistic tracker identification would enable the adversary to track down any of those trackers, respectively steal the attached item. To give an example, where a Galaxy SmartTag+ represents a holistically identified tracker since the Galaxy SmartTag+ is regarded as easily identifiable. An adversary with an app that scans for a Galaxy SmartTag+ among BLE advertisements, can enter a room and check if there is such a tracker present. If this

is the case and the owner is not present, the adversary can simply search the room for the tracker and probably find a valuable item attached to it. Also, the owner would not notice that the item has been stolen. Extending this scenario, it can be even easier for an adversary to locate an item. Some trackers are capable of playing sound or can be found by using an AR compass with the help of UWB technology. This makes it even easier for an adversary to locate an item.

To summarize the discussion about tracker identification, easily identifiable trackers have the benefit to protect victims from stalking attacks in a fast manner. Not much work is needed to identify such a tracker and hence protect the user. On the other side, easily identifiable trackers pose security concerns, as they can be located without heavily studying their BLE advertisements. Thus, an item attached to such a tracker is more easy to track down for an adversary.

6.2 Scanning Behaviour

For this work, an app has been built, which is described in Section 4.3. The way its services have been designed is not heavily based on scientific research. Furthermore, as mentioned in the experiments regarding the occurrences in Section 5.2.4, the design of the `BluetoothScanningService` might restrict the ability of the occurrences parameter to classify malicious from non-malicious trackers. Hence, this service is discussed in more detail.

In the app, two values define the scanning behaviour. One is the `INTERVAL_BLE_SCAN` and the other is the `SCAN_PERIOD`. The `INTERVAL_BLE_SCAN` is set to 18 seconds, whereas the `SCAN_PERIOD` is set to 12 seconds. The reason for the decision on those values is found in the explanation regarding the `BluetoothScanningService` in Section 4.3.3. With those values and within a full minute, the app scans in total for 24 seconds, at two distinct time windows as it can be seen in Figure 32 a). Furthermore, with this configuration, the app always scans in the identical time windows of a minute. That is from second 0 to second 12 and from second 30 to second 42. In the following minute, scanning is carried out during the same time windows. Moreover, this scanning behaviour only allows getting two BLE advertisements of a BLE device within a minute.

Changing the value for the `INTERVAL_BLE_SCAN`, a different scanning behaviour can be achieved. Changing the `SCAN_PERIOD` is not recommended, as it would become likely that advertisements for some BLE devices might not be scanned at all during a scan period due to the allowed advertisement interval of 10.24 seconds [27]. Nonetheless, adapting the `INTERVAL_BLE_SCAN` results in a different scan behaviour, as seen in Figure 32 b). In this configuration, the `INTERVAL_BLE_SCAN` is set to 12 seconds. Hence, within the first minute, the app would scan from second 0 to 12, 24 to 36, and 48 to 60. This results in an additional scan of BLE advertisements. Moreover, within the second minute, the app would scan from second 12 to 24 and from second 36 to 48. In the second minute, the number of scanned BLE advertisements remains the same compared to the established configuration. Nonetheless, one more interesting behaviour can be observed, as the time windows of the scan periods alternate.

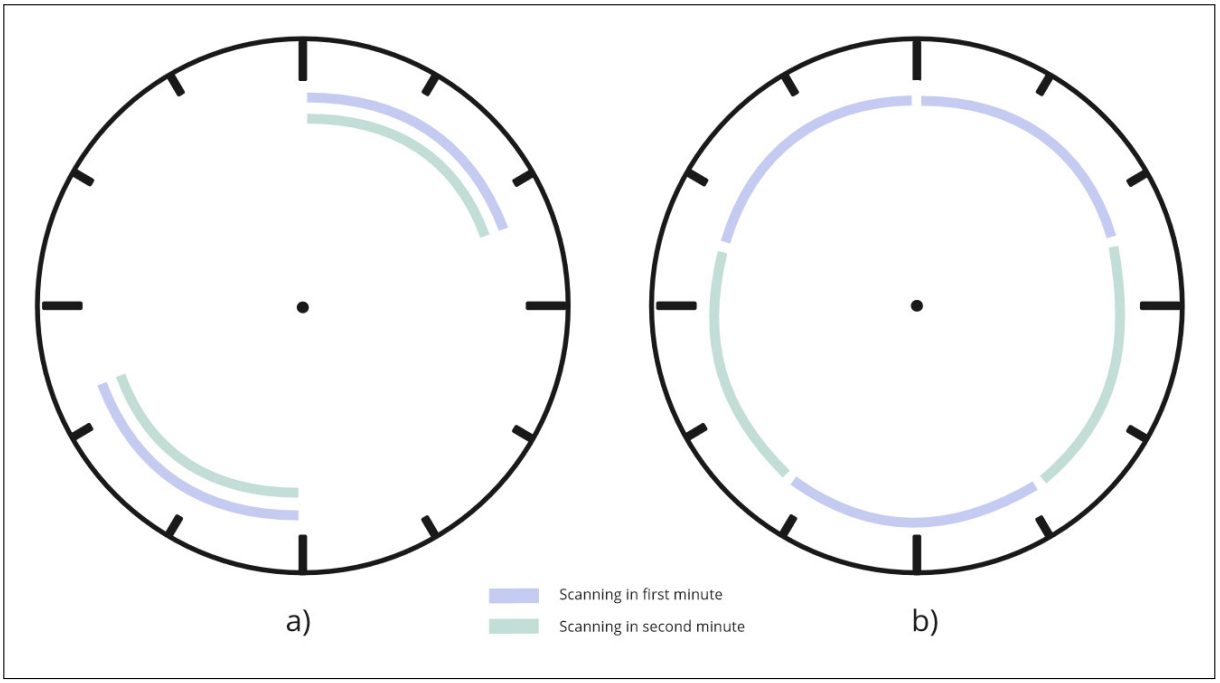

Figure 32: Visualizing a scan period of 12 seconds with different scan intervals over a total time period of two minutes. In a) the scan interval is set to 18 seconds. In b) the scan interval is set to 12 seconds.

To conclude, other values for the `INTERVAL_BLE_SCAN` and `SCAN_PERIOD` might be chosen. Depending on those values, the scanning behaviour changes. This has an impact on how to treat the occurrences parameter within the tracking algorithm. Moreover, the distance parameter might also be affected by this new scanning behaviour. For example, consider the user making a turn in between two scans of a BLE device. The computed distance might be smaller compared to the actual distance covered. This would be improved by an additional scan between the existing scans. Finally, the time parameter would not be affected by this new scanning behaviour. The measured time between the two existing scans would not change if an additional scan occurs between them.

6.3 False Positives of Tracking Protection

First of all, the tracking algorithm shows promising results as described in Section 5.2.4. The main insight is, that the distance parameter appears to be the leading parameter regarding a decision on whether a BLE device should be classified as a malicious or non-malicious tracker. Figure 33 shows how false positives and false negatives were eliminated by increasing the value for the distance parameter. Hence, a user travelling by him- or herself, gets notified about malicious trackers appropriately. It is assumed that this can be transferred to a scenario where a user is driving a car, bicycle or other vehicles, provided they are travelling alone. The only difference is, that the user travels faster, thus the threshold of the distance parameter is exceeded faster in time.

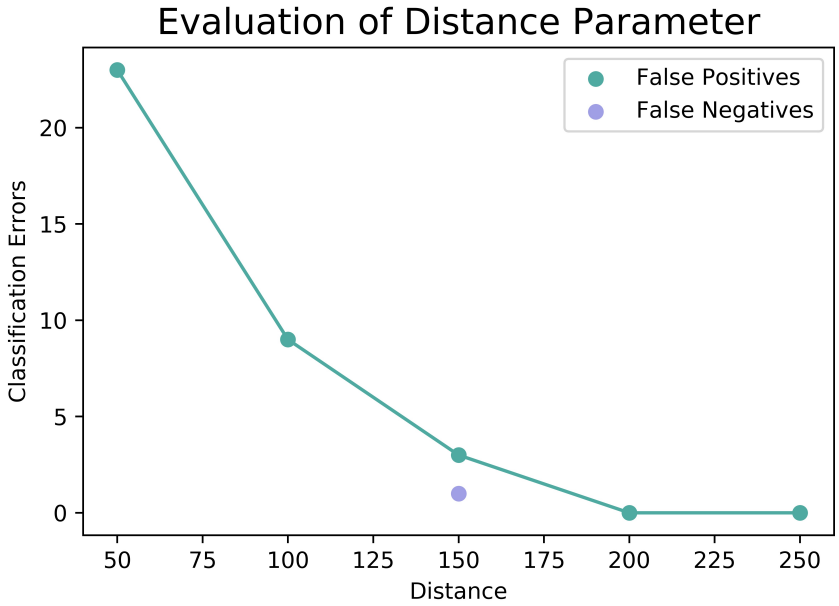

Figure 33: Graph showing the performance of the tracking algorithm where the distance parameter has been increased iteratively. The occurrences and the time parameter were set to their minimal values which is 2 occurrences and 1 minute.

However, a user of the app, travelling with a friend or surrounded by people will receive false positives. For example, assuming a scenario where the user travels with a friend, who also carries a BLE-capable device such as a smartphone. In this scenario, the tracking protection will not be able to realize that the user is travelling with a friend. Hence, the thresholds of the tracking algorithm will most likely be exceeded due to the carried along BLE device of the friend and therefore a false positive occurs. The same is true for a user travelling on public transportation. Usually, other people are present in public transport, which probably carry some BLE-capable devices. Here, the app is also not capable of realizing that the user is surrounded by people. Hence, false positives can not be avoided with the given selection of the values for the tracking algorithm.

Moreover, by the behaviour of the `LocationTrackingService` which is explained in Figure 19 cases can exist where a user receives false positives. This is due to the fact, that this service evaluates how long the distance is that the user has travelled, whereas the direction of travelling is not considered. For example, a user can walk back and forth within a distance of 10 meters. At some point, the `LocationTrackingService` considers the user to be moving. Thus, if a BLE-capable device is near, it might be classified as a malicious tracker. The reason for this is, that the tracking protection can only use the location received from the app. It is not possible to get the actual location of a tracker which is not moving in this scenario. Hence, as the user keeps walking back and forth, the BLE device is regarded as travelling with the user and at some point, the thresholds of the tracking algorithm will be exceeded and a false positive occurs.

Finally, within the experiments regarding the distance parameter as described in Section 5.2.4, a false positive has occurred, where another person crossed the way of the user. This has been further evaluated and the occurrences parameter has been raised to

4 occurrences, such that this false positive is eliminated. Nonetheless, with those configurations, considering a scenario where another person would cross the user 4 times, a false positive would still occur.

6.4 Mac Randomization

All trackers used in this work have been analyzed regarding their MAC randomization pattern. The AirTag, Chipolo ONE Spot and the Tile tracker do not randomize their MAC address while they are not in connection range with their corresponding owner device. At least for one hour, this is true. The Galaxy SmartTag+ on the other hand randomizes its MAC address no matter if it is close to the owner or not.

Furthermore, detecting a malicious tracker is based on its MAC address. This means the tracker algorithm decides if a tracker is malicious or non-malicious based on all BLE advertisements mapped to the same MAC address. Hence, if a BLE device randomizes its MAC address, it is regarded as a new BLE device, respectively tracker. In the case of a Galaxy SmartTag+, which randomizes its MAC address every 5 to 10 minutes, a false positive occurs due to this fact. Users for example travelling with a Galaxy SmartTag+ for over 10 minutes will receive two notifications about two different Galaxy SmartTag+ following them, as the tracker has changed its MAC address. This will also be true for other devices that randomize their MAC address and are used to track a user.

6.5 RSSI as a shield

It seems, the basic analysis of the RSSI values only states that the RSSI value decreases the further away a tracker is. However, while working on this topic, the idea arose to include the RSSI value in the`BluetoothScanningService`. This general decline of the RSSI value could be used as a shield for trackers which are far away. In other words, the experiments showed that the trackers have an RSSI value of around -75 if they are 4 meters away and a value of around -90 if they are 10 meters away. This information can be used to filter for example trackers which have a lower RSSI value than -90. It is regarded as very unlikely, that a BLE device is tracking a victim with an RSSI value of -90, respectively a tracker which is about 10 meters away from the victim. Nonetheless, the experiments regarding the RSSI values have only been conducted with BLE trackers. Hence, this cannot be transferred directly to any BLE device. It might be that non-BLE trackers, which are 10 meters away from the receiver, might broadcast their advertisements with a higher or lower RSSI value.

Chapter 7

Conclusion and Future Work

This last chapter concludes this work. Additionally, recommended future work is pointed out in the field of tracking protection. Moreover, the section Future Work includes app improvements for developers continuing this work.

7.1 Conclusion

This work aimed to protect Android users from stalking attacks using BLE devices in general. To achieve this, it has been assessed, if BLE trackers can be identified and distinguished from BLE devices. From the perspective of Android's BLE API, it has been shown that it is not possible. This addresses contribution (i). However, studying the BLE advertisements of the Galaxy SmartTag+ showed, that it is possible to map its advertisement, respectively that it is possible to determine that the advertisement is emitted by a Galaxy SmartTag+. This is also true for the AirTag, Chipolo ONE Spot and Tile trackers which have been identified in previous work from Heinrich et al. [24]. Therefore, it is likely that BLE advertisements from not yet-identified trackers can be assigned to the respective trackers. However, this needs to be accomplished for every tracker individually.

Furthermore, an app has been built with an adaptable tracking algorithm to detect malicious trackers. This addresses contribution (ii). The app represents a prototype and serves as a basis to fully protect users from stalking attacks. A scenario where a user walks from a starting point to a destination can be covered with the app such that the user gets notified about malicious trackers as early as possible. It is assumed that the app covers scenarios where a user drives in a car, rides a bike, or any other vehicle as long as he or she is travelling alone. This is due to the fact, that in those scenarios the user of the app simply travels with a faster velocity.

Additionally, the app has been used to study the parameters of the implemented tracking algorithm that decides whether a BLE device is a malicious tracker or not. The aim of the tracking algorithm is to inform the user as early as possible about malicious

trackers. Within the given scenario, where the user walked a distance of around 500 meters in around 5 minutes the tracking algorithm is capable of detecting and notifying of malicious trackers. The tracking algorithm shows promising results setting the distance parameter to 200 meters, the time parameter to 1 minute, and the occurrences parameter to 4 occurrences. This addresses contribution (iii). Furthermore, it could be identified, that the distance parameter is the leading parameter to identify malicious trackers. The occurrences parameter supports the distance parameter such that the classification of the tracking algorithm improves. The time parameter is so far considered to be a restricting parameter, as it delays the point in time when a notification about a malicious tracker is sent to the user.

Regarding secondary contributions of this work, tracking protection still faces the problem of BLE devices that randomize their MAC address. Those devices will be classified as additional malicious trackers, as soon as they randomize their MAC address. This is definitely the case for the Galaxy SmartTag+.

Furthermore, the investigation of the RSSI values shows, that the RSSI value cannot be used to distinguish between trackers. However, the RSSI values might be able to enhance tracking protection on a different level. These values could be used as a shield to filter respectively ignore signals from BLE devices that are considered too far away to act as stalking ware.

To summarize this work, the developed Android app equips users with tracking protection. The tracking algorithm of the app can be adapted by the user. Once such a device respectively a malicious tracker is detected according to the tracking algorithm, the app notifies the user instantly. Where possible, the app states which type of tracker has been detected due to the implemented tracker identification. Furthermore, with the adaptable tracking algorithm, several experiments have been conducted to analyze the performance of the tracking algorithm. With the values declared above, the tracking algorithm shows promising results. Selecting those parameters allows notifying users as early as possible about malicious trackers. This eventually prevents users from stalking attacks. In a nutshell, this work and the associated app build a basis for future work which aims at enhancing tracking protection, such that users are fully protected from stalking attacks with BLE devices.

7.2 Future Work

The app built for this work is a prototype, hence it is not ready for the market. It can be used to protect users from stalking attacks, but further improvements are necessary. The most important issue is to allow the app to work while the phone is locked, respectively in doze mode. At this point, the app only works while the phone is awake so to speak. Thus, a user would need to make sure all the time that the phone remains unlocked. This is not manageable. Nevertheless, this can be solved by properly implementing a `ACTION_REQUEST_IGNORE_BATTERY_OPTIMIZATIONS`. Afterwards, the app would need to be tested, if it works as intended.

The experiments regarding the tracking algorithm showed good results. However, the identified values for the tracking algorithm are related to a scenario, in which a user walks from a starting point to a destination by himself or herself. Other scenarios, where a user for example travels with a friend or by public transportation, will lead to false notifications about malicious trackers. One idea to solve this is, that the app can detect in which scenario it is. This means the app would realize for example that the user travels with a friend. Hence, recurring BLE advertisements from the devices a friend might carry along could be filtered or the values for the tracking parameters would need to be adapted. Another idea that can be pursued is studying the time parameter in more depth. For example, in a scenario where the user travels by public transportation, the time parameter might be of use to avoid false notifications. The assumption is, that the time spent on public transportation is limited. Hence, the time parameter might be able to cover this time period. Thus, it is recommended to further investigate in this direction.

In this work, a trade-off has been identified regarding holistic tracker identification. As described in Section 5.3 this trade-off is between how easily a tracker can be identified versus the security concerns it brings along. Generally, a holistic tracker identification needs to address this trade-off. It needs to be designed such that trackers can be identified easily. On the other hand, it needs to be complex enough such that the security of an owner is guaranteed, respectively the owner's item cannot be stolen without much effort. Hence, a solution to this trade-off depends on how holistic tracker identification is achieved. A first idea is to use artificial intelligence to identify trackers based on their BLE advertisements. This way, a pattern might be found, which has not yet been discovered, to classify BLE advertisements into advertisements coming from a BLE tracker or BLE device. In case such a holistic tracker classification based on artificial intelligence is possible, the question is how this solution addresses the security concerns. At this point, it is assumed, that the benefit of this solution might lie in the nature of artificial intelligence. A model which classifies BLE trackers versus BLE devices is based on a lot of labelled training and test data. Gathering this data results in a lot of work concerning time. Moreover, knowledge is needed in this area to create such a model. Hence, an adversary trying to steal items attached to trackers would need to invest time and knowledge to create such a model. This might not be worth the effort. However, this is only a barrier with respect to time.

Regarding the secondary contributions, one problem that remains is devices that randomize their MAC address. Those devices will generate false notifications about malicious trackers as soon as they change their MAC address and exceed the thresholds of the tracking algorithm. Overcoming this problem will improve the app from the user's perspective. Having two notifications of the same tracker does not harm the user, but might give the user the impression that the app does not work properly. In the case of the Galaxy SmartTag+, the data retrieved from the service UUID might be a starting point. This data could be held against the data from a second Galaxy SmatTag+ tracker to see if eventually the advertisement can be mapped to the corresponding tracker.

Finally, the experiments regarding the RSSI values assured, that the RSSI value of a tracker decreases the further away the tracker is from the receiver. This behaviour can be used as a shield. Trackers that are regarded to be too far away from the victim do not have to be considered by the tracking algorithm. In other words, a tracker with an RSSI value of -90 dBm is approximately 10 meters away. Such a tracker is most likely not used as

stalking ware. However, it needs to be evaluated at which distance respectively at which RSSI value a tracker is to be considered too far away. Moreover, those RSSI experiments are all based on actual BLE trackers. Whether BLE devices show the same behaviour can not be answered at this point. Therefore, experiments to evaluate the appropriate RSSI threshold to filter signals are recommended to be conducted. Furthermore, analysing the behaviour of the RSSI values of BLE devices should be included in those experiments.

Bibliography

[1] "Bluetooth Technology Overview | Bluetooth® Technology Website," [Online]. Available: https://www.bluetooth.com/learn-about-bluetooth/tech-overview/, *last accessed: 20/12/2022*.

[2] "Use the Find My app to locate a missing device or item - Apple Support," [Online]. Available: https://support.apple.com/en-us/HT210515, *last accessed: 09/12/2022*.

[3] "Airtag-Stalking: Hat Apple die Gefahr unterschätzt?" [Online]. Available: https://www.nzz.ch/panorama/airtag-stalking-hat-apple-die-gefahr-unterschaetzt-ld.1688765?reduced=true, *last accessed: 09/12/2022*.

[4] "Apple AirTags are 'dangerous' device used by stalkers, lawsuit alleges," [Online]. Available: https://eu.usatoday.com/story/tech/news/2022/12/06/apple-airtags-stalking-lawsuit/10843997002/, *last accessed: 09/12/2022*.

[5] "Stalking-Werkzeug: Bernerin (19) findet fremden Airtag unter ihrem Beifahrersitz - "Ich geriet in Panik" - 20 Minuten," [Online]. Available: https://www.20min.ch/story/ich-hatte-panik-bernerin-19-findet-fremden-airtag-unter-ihrem-beifahrersitz-460234987350, *last accessed: 09/12/2022*.

[6] "How thieves use Apple AirTags to steal high-end cars and track them with their iPhones | The US Sun," [Online]. Available: https://www.the-sun.com/tech/4225369/apple-airtags-track-cars-theft/, *last accessed: 09/12/2022*.

[7] "What to do if you get an alert that an AirTag, Find My network accessory, or set of AirPods is with you - Apple Support," [Online]. Available: https://support.apple.com/en-us/HT212227, *last accessed: 09/12/2022*.

[8] "Tracker Detect - Apps on Google Play," [Online]. Available: https://play.google.com/store/apps/details?id=com.apple.trackerdetect&hl=en&gl=US, *last accessed: 09/09/2022*.

[9] "AirGuard - AirTag Schutz - Apps bei Google Play," [Online]. Available: https://play.google.com/store/apps/details?id=de.seemoo.at_tracking_detection.release&gl=US, *last accessed: 15/10/2022*.

[10] E. D. Ayele, N. Meratnia, and P. J. M. Havinga, "An asynchronous dual radio opportunistic beacon network protocol for wildlife monitoring system," in *2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, IEEE, 2019, pp. 1–7, ISBN: 1728115426.

[11] Y. Zhuang, J. Yang, Y. Li, L. Qi, and N. El-Sheimy, "Smartphone-Based Indoor Localization with Bluetooth Low Energy Beacons," *Sensors 2016*, vol. 16, p. 596, Apr. 2016, ISSN: 1424-8220. DOI: 10.3390/S16050596. [Online]. Available: https://www.mdpi.com/1424-8220/16/5/596/htm.

[12] P. Kriz, F. Maly, and T. Kozel, "Improving Indoor Localization Using Bluetooth Low Energy Beacons," *Mobile Information Systems, Vol. 2016*, 2016, ISSN: 1875905X. DOI: 10.1155/2016/2083094.

[13] R. Giuliano, G. C. Cardarilli, C. Cesarini, *et al.*, "Indoor Localization System Based on Bluetooth Low Energy for Museum Applications," *Electronics 2020*, vol. 9, p. 1055, Jun. 2020, ISSN: 2079-9292. DOI: 10.3390/ELECTRONICS9061055. [Online]. Available: https://www.mdpi.com/2079-9292/9/6/1055/htm%20https://www.mdpi.com/2079-9292/9/6/1055.

[14] S. Chai, R. An, and Z. Du, "An Indoor Positioning Algorithm using Bluetooth Low Energy RSSI," pp. 274–276, Apr. 2016, ISSN: 2352-5401. DOI: 10.2991/AMSEE-16.2016.72. [Online]. Available: https://www.atlantis-press.com/proceedings/amsee-16/25858154.

[15] A. Mussina and S. Aubakirov, "RSSI Based Bluetooth Low Energy Indoor Positioning," *IEEE 12th International Conference on Application of Information and Communication Technologies, AICT 2018 - Proceedings*, Oct. 2018. DOI: 10.1109/ICAICT.2018.8747020.

[16] S. Sadowski and P. Spachos, "RSSI-Based Indoor Localization with the Internet of Things," *IEEE Access*, vol. 6, pp. 30 149–30 161, Jun. 2018, ISSN: 21693536. DOI: 10.1109/ACCESS.2018.2843325.

[17] Z. Jianyong, L. Haiyong, C. Zili, and L. Zhaohui, "RSSI based Bluetooth low energy indoor positioning," *IPIN 2014 - 2014 International Conference on Indoor Positioning and Indoor Navigation*, pp. 526–533, 2014. DOI: 10.1109/IPIN.2014.7275525.

[18] X. Li, D. Wei, Q. Lai, Y. Xu, and H. Yuan, "Smartphone-based integrated PDR/GPS/Bluetooth pedestrian location," *Advances in Space Research*, vol. 59, no. 3, pp. 877–887, Feb. 2017, ISSN: 0273-1177. DOI: 10.1016/J.ASR.2016.09.010.

[19] A. Heinrich, M. Stute, T. Kornhuber, and M. Hollick, "Who Can Find My Devices? Security and Privacy of Apple's Crowd-Sourced Bluetooth Location Tracking System," *Proceedings on Privacy Enhancing Technologies*, pp. 227–245, Jul. 2021. DOI: 10.2478/POPETS-2021-0045.

[20] M. Nikodem and M. Bawiec, "Experimental evaluation of advertisement-based bluetooth low energy communication," *Sensors*, vol. 20, no. 1, p. 107, 2019, ISSN: 1424-8220.

[21] J. K. Becker, D. Li, and D. Starobinski, "Tracking Anonymized Bluetooth Devices," *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 3, pp. 50–65, Jul. 2019. DOI: https://doi.org/10.2478/popets-2019-0036.

[22] A. Heinrich, M. Stute, and M. Hollick, "OpenHaystack: A framework for tracking personal bluetooth devices via Apple's massive find my network," *WiSec 2021 - Proceedings of the 14th ACM Conference on Security and Privacy in*

Wireless and Mobile Networks, pp. 374–376, Jun. 2021. DOI: 10.1145/3448300.3468251.

[23] "iCloud - Find My - Apple," [Online]. Available: https://www.apple.com/icloud/find-my/, *last accessed: 29/09/2022*.

[24] A. Heinrich, N. Bittner, and M. 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.

[25] "The Bluetooth® Low Energy Primer | Bluetooth® Technology Website," [Online]. Available: https://www.bluetooth.com/bluetooth-resources/the-bluetooth-low-energy-primer/?utm_source=internal&utm_medium=blog&utm_campaign=technical&utm_content=the-bluetooth-low-energy-primer, *last accessed: 08/09/2022*.

[26] "Assigned Numbers | Bluetooth® Technology Website," [Online]. Available: https://www.bluetooth.com/specifications/assigned-numbers/, *last accessed: 03/11/2022*.

[27] "Ellisys Bluetooth Video 3: Advertisements - YouTube," [Online]. Available: https://www.youtube.com/watch?v=be9ct7OKI7s, *last accessed 16/10/2022*.

[28] G. Celosia, M. Cunche, and U. Lyon, "Discontinued Privacy: Personal Data Leaks in Apple Bluetooth-Low-Energy Continuity Protocols," *Proceedings on Privacy Enhancing Technologies*, vol. 2020, no. 1, pp. 26–46, Jan. 2020, ISSN: 2299-0984. DOI: 10.2478/POPETS-2020-0003. [Online]. Available: https://www.bluetooth.com/specifications/assigned-.

[29] "Core Specification - Bluetooth® Technology Website," [Online]. Available: https://www.bluetooth.com/specifications/specs/core-specification-5-3/, *last accessed: 08/12/2022*.

[30] "Find My Network Accessory Specification Developer Preview Release R1," [Online]. Available: https://images.frandroid.com/wp-content/uploads/2020/06/Find_My_network_accessory_protocol_specification.pdf, *last accessed: 02/01/2023*.

[31] "AirTag - Apple," [Online]. Available: https://www.apple.com/airtag/, *last accessed: 08/09/2022*.

[32] "Finde deine Schlüssel, Portemonnaie und Telefon - Chipolo," [Online]. Available: https://chipolo.net/de/, *last accessed: 10/10/2022*.

[33] "Galaxy SmartTag Black | Bluetooth Tracker | Samsung Schweiz," [Online]. Available: https://www.samsung.com/ch/mobile-accessories/galaxy-smarttag-black-ei-t5300bbegeu/, *last accessed: 16/10/2022*.

[34] "Learn How Tile's Bluetooth Tracking Device & Tracker App Helps You Find Your Lost Things | Tile," [Online]. Available: https://ch.tile.com/en/how-it-works, *last accessed: 03/11/2022*.

[35] "Find Your Keys, Wallet & Phone with Tile's App and Bluetooth Tracker Device | Tile," [Online]. Available: https://ch.tile.com/en, *last accessed: 03/11/2022*.

[36] "Pro 2-pack | Tile," [Online]. Available: https://ch.tile.com/en/product/686612/pro-2-pack, *last accessed: 03/11/2022*.

[37] "Create a project | Android Developers," [Online]. Available: https://developer.android.com/studio/projects/create-project, *last accessed: 24/12/2022.*

[38] "Android 8.0 Features and APIs | Android Developers," [Online]. Available: https://developer.android.com/about/versions/oreo/android-8.0, *last accessed: 24/12/2022.*

[39] "UI layer | Android Developers," [Online]. Available: https://developer.android.com/topic/architecture/ui-layer, *last accessed: 02/12/2022.*

[40] "Data layer | Android Developers," [Online]. Available: https://developer.android.com/topic/architecture/data-layer, *last accessed: 02/12/2022.*

[41] "Repository Pattern," [Online]. Available: https://developer.android.com/codelabs/basic-android-kotlin-training-repository-pattern#0, *last accessed: 03/12/2022.*

[42] "Optimize for Doze and App Standby | Android Developers," [Online]. Available: https://developer.android.com/training/monitoring-device-state/doze-standby, *last accessed: 03/12/2022.*

[43] "Settings | Android Developers," [Online]. Available: https://developer.android.com/reference/android/provider/Settings#ACTION_REQUEST_IGNORE_BATTERY_OPTIMIZATIONS, *last accessed: 03/12/2022.*

[44] "LocationRequest | Google Play services | Google Developers," [Online]. Available: https://developers.google.com/android/reference/com/google/android/gms/location/LocationRequest#getIntervalMillis(), *last accessed: 03/12/2022.*

[45] "Find BLE devices | Android Developers," [Online]. Available: https://developer.android.com/guide/topics/connectivity/bluetooth/find-ble-devices, *last accessed: 05/12/2022.*

[46] "Learn Kotlin - lateinit vs lazy," [Online]. Available: https://blog.mindorks.com/learn-kotlin-lateinit-vs-lazy, *last accessed: 18/10/2022.*

[47] "ScanSettings | Android Developers," [Online]. Available: https://developer.android.com/reference/android/bluetooth/le/ScanSettings, *last accessed: 20/12/2022.*

[48] "AirGuard/DeviceManager.kt at main · seemoo-lab/AirGuard · GitHub," [Online]. Available: https://github.com/seemoo-lab/AirGuard/blob/main/app/src/main/java/de/seemoo/at_tracking_detection/database/models/device/DeviceManager.kt, *last accessed: 10/10/2022.*

[49] "Kotlin Bitwise and Bitshift Operations (With Examples)," [Online]. Available: https://www.programiz.com/kotlin-programming/bitwise, *last accessed: 03/11/2022.*

[50] "AirGuard/Tile.kt at main · seemoo-lab/AirGuard," [Online]. Available: https://github.com/seemoo-lab/AirGuard/blob/main/app/src/main/java/de/seemoo/at_tracking_detection/database/models/device/types/Tile.kt, *last accessed: 08/12/2022.*

[51] "Pendlermobilität | Bundesamt für Statistik," [Online]. Available: `https://www.bfs.admin.ch/bfs/de/home/statistiken/mobilitaet-verkehr/personenverkehr/pendlermobilitaet.html`, *last accessed: 27/10/2022*.

[52] *Arbeitsvolumenstatistik (AVOL)*, DE. Bundesamt für Statistik (BFS), Nov. 2021, p. 10. [Online]. Available: `https://dam-api.bfs.admin.ch/hub/api/dam/assets/19904368/master`.

[53] "Understanding Bluetooth Range | Bluetooth® Technology Website," [Online]. Available: `https://www.bluetooth.com/learn-about-bluetooth/key-attributes/range/`, *last accessed: 21/10/2022*.

Abbreviations

ADVB	Advertising Broadcast
AoA	Angle of Arrival
AoD	Angle of Departure
API	Application Programming Interface
APK	Android Application Package
AR	Augmented Reality
BER	Bit Error Rate
BLE	Bluetooth Low Energy
CRC	Cyclic Redundancy Check
DAO	Data Access Object
dBm	decibel-milliwatts
FEC	Forward Error Correction
GFSK	Gaussian Frequency Shift Keying
GPS	Global Positioning System
HCI	Host Controller Interface
LSB	Least Significant Bit
MAC	Media Access Control
MSB	Most Significant Bit
MVVM	Model View ViewModel
NFC	Near Field Communication
OSI	Open Systems Interconnection
PDU	Protocol Data Unit
RSSI	Received Signal Strength Indicator
SDK	Software Development Kit
SIG	Special Interest Group
TDD	Time Division Duplex
Tx	Transaction
UI	User Interface
UUID	Universally Unique Identifier
UWB	Ultra-wideband

Glossary

BLE device Any device which has BLE capabilities.

BLE tracker A BLE device with the intended purpose of tracking and locating an item.

Client/Server Bluetooth follows a Client/Server pattern when two devices communicate with each other. A server stores data and a client operates with data from the server. In this work, a server is regarded as a tracking device and a client is a smartphone trying to detect such a tracking device.

Connectable PDU Types of advertisements can be connectable. This means that a device receiving may connect to the advertising device.

MAC Address Media Access Control Address of a device.

Malicious Tracker A tracker which has been attached by an adversary to a victim in order to track the position of the victim, without the victim knowing it.

Scannable PDU Types of advertisements can be scannable. This means that a device receiving this particular advertisement can respond with a scan request PDU to get more advertising data.

Scan Period Number of seconds during which the app scans for Bluetooth Low Energy devices.

Scan Interval Number of seconds between two scan periods, respectively where the app is not scanning for BLE devices.

Tracker classification This term is used to classify an identified tracker into either a malicious tracker or a non-malicious tracker.

Tracker identification This term is used to determine the type of tracker. That is either an AirTag, Chipolo ONE Spot, Galaxy SmartTag+ or a Tile tracker.

List of Figures

List of Tables

Appendix A

Android Bluetooth API records

The following sections show for every tracker used in this work an example of a `ScanResult`[21], a `ScanRecord`[22], and a `BluetoothDevice`[23]. The following results have been recorded during the advertisement state of the BLE Trackers. This means, the tracker has lost connection to its owner and is now advertising or broadcasting its presence.

[21] `https://developer.android.com/reference/android/bluetooth/le/ScanResult`
[22] `https://developer.android.com/reference/android/bluetooth/le/ScanRecord`
[23] `https://developer.android.com/reference/android/bluetooth/BluetoothDevice`

A.1 AirTag

Example ScanResult of an AirTag		
Method	**Value**	**Description**
describeContents()	0	Describe the kinds of special objects contained in this Parcelable instance's marshaled representation.
getAdvertisingSid()	255	Returns the advertsing id SID_NOT_PRESENT = 255
getDataStauts()	0	Returns the data Status DATA_COMPLETE = 0 DATA_TRUNCATED = 2
getDevice()	see: Example BluetoothDevice for an AirTag	Returns remote BluetoothDevice
getPeriodicAdvertisingInterval()	0	Returns the periodic advertising interval in units of 1.25ms between 6 (7.5ms) and 65536 (81918.75ms) PERIODIC_INTERVAL_NOT_PRESENT = 0
getPrimaryPhy()	1	Returns the primary Physical Layer of the advertisement PHY_LE_1M = 1 PHY_LE_CODED = 3
getRssi()	-47	Returns the received signal strength in dBm within the range of [-127, 126]
getScanRecord()	see: Example ScanRecord for an AirTag	Returns an instance of a ScanRecord
getSecondaryPhy()	0	Returns the secondary Physical Layer of the advertisement PHY_LE_1M = 1, PHY_LE_2M = 2 PHY_LE_CODED = 3, PHY_UNUSED = 0
...		

...		
getTimestampNanos()	3287669350801853	Returns timestamp since boot when the scan record was observed.
getTxPower()	127	Returns the transmit power in dBm [-127,126] TX_POWER_NOT_PRESENT = 127
hashCode()	772993152	Returns a hashcode value for the current object
isConnectable()	true	Returns boolean if this object represents a connectable scanr esult
isLegacy()	true	Returns true if this object represents a legacy scan result. Those do not contain advanced advertising information as specified in Bluetooth Core Specification v5.
toString()	ScanResult{ device=DD:CB:15:04:4D:26, scanRecord=ScanRecord[mAdvertiseFlags=-1, mServiceUuids=null, mServiceSolicitationUuids=[], mManufacturerSpecificData={76 =[18, 25, 16, 108, -28, 119, -73, 39, -89, 125, 29, -23, 107, -73, -78, -61, 10, -112, -36, 39, 64, 79, -86, 22, 93, 1, 38]}, mServiceData={}, mTxPowerLevel=-2147483648, mDeviceName=null, mTDSData=null], rssi=-47, timestampNanos=3287669350801853, eventType=27, primaryPhy=1, secondaryPhy=0, advertisingSid=255, txPower=127, periodicAdvertisingInterval=0}	String Representation of object
The description of the functions is based on: https://developer.android.com/reference/android/bluetooth/le/ScanResult Not included Functions and why: • equals(): comparing ScanResults while analyzing ScanResults was neither possible nor necessary		

Figure 34: Appendix: ScanResult of an AirTag.

Example ScanRecord of an AirTag		
Method	**Value**	**Description**
getAdvertiseFlags()	-1	Returns the advertising flags indicating the discoverable mode and capability of the device. Returns -1 if the flag field is not set.
getBytes()	[B@e6bafaf	Returns raw bytes of scan record.
getDeviceName()	null	Returns the local name of the BLE device.
getManufacturerSpecificData()	{76=[B@3bb4fbc}	Returns a sparse array of manufacturer identifier and its corresponding manufacturer specific data.
getServiceData()	{}	Returns the service data byte array associated with the serviceUuid
getServiceSolicitationUuids()	[]	Returns a list of service solicitation UUIDs within the advertisement that are used to identify the Bluetooth GATT services.
getServiceUuids()	null	Returns a list of service UUIDs within the advertisement that are used to identify the bluetooth GATT services.
...		

...		
getTxPowerLevel()	-2147483648	Returns the transmission power level of the packet in dBm. Returns -2147483648 if the field is not set. This value can be used to calculate the path loss of a received packet using the following equation: pathloss = txPowerLevel - rssi
toString()	ScanRecord [mAdvertiseFlags=-1, mServiceUuids=null, mServiceSolicitationUuids=[], mManufacturerSpecificData={76 =[18, 25, 16, 108, -28, 119, -73, 39, -89, 125, 29, -23, 107, -73, -78, -61, 10, -112, -36, 39, 64, 79, -86, 22, 93, 1, 38]}, mServiceData={}, mTxPowerLevel=-2147483648, mDeviceName=null, mTDSData=null]	String Representation of object
The description of the functions is based on: https://developer.android.com/reference/android/bluetooth/le/ScanRecord Not included Functions and why: • getAdvertisingDataMap(): has been added in API level 33		

Figure 35: Appendix: ScanRecord of an AirTag.

Example BluetoothDevice of an AirTag		
Method	**Value**	**Description**
fetchUuidsWithSdp()	true	Perform a service discovery on the remote device to get the UUIDs supported. Returns a boolean.
getAddress()	DD:CB:15:04:4D:26	Returns the hardware address of this BluetoothDevice.
getAlias()	null	Get the locally modifiable name (alias) of the remote Bluetooth device.
getBluetoothClass()	0	return Bluetooth Class of Remote device PROFILE_HEADSET = 0
getBondState()	10	Get the bond state of the remote device. BONDE_NONE = 10 BOND_BONDING = 11 BOND_BONDED = 12
getName()	null	Get the friendly Bluetooth name of the remote device.
...		

...		
getType()	0	Get the Bluetooth device type of the remote device. DEVICE_TYPE_CLASSIC = 1 DEVICE_TYPE_LE = 2 DEVICE_TYPE_DUAL = 3 DEVICE_TYPE_UNKNOWN = 0
getUuids()	null	Returns the supported features (UUIDs) of the remote device.
toString()	DD:CB:15:04:4D:26	string representation of object
The description of the functions is based on: https://developer.android.com/reference/android/bluetooth/BluetoothDevice Not included Functions and why: • connectGatt() up to equals(): no connection was in scope as well comparing to other device • hashCode(): creates a hash of the object which for this task is of no further use • setAliats() up to setPin(): setters are of no help analyzing a bluetooth device		

Figure 36: Appendix: BluetoothDevice of an AirTag.

A.2 Chipolo One Spot

Example ScanResult of a Chipolo ONE Spot		
Method	**Value**	**Description**
describeContents()	0	Describe the kinds of special objects contained in this Parcelable instance's marshaled representation.
getAdvertisingSid()	255	Returns the advertsing id SID_NOT_PRESENT = 255
getDataStauts()	0	Returns the data Status DATA_COMPLETE = 0 DATA_TRUNCATED = 2
getDevice()	see: Example BluetoothDevice for a Chipolo ONE Spot	Returns remote BluetoothDevice
getPeriodicAdvertisingInterval()	0	Returns the periodic advertising interval in units of 1.25ms between 6 (7.5ms) and 65536 (81918.75ms) PERIODIC_INTERVAL_NOT_PRESENT = 0
getPrimaryPhy()	1	Returns the primary Physical Layer of the advertisement PHY_LE_1M = 1 PHY_LE_CODED = 3
getRssi()	-50	Returns the received signal strength in dBm within the range of [-127, 126]
getScanRecord()	see: Example ScanRecord for a Chipolo ONE Spot	Returns an instance of a ScanRecord
getSecondaryPhy()	0	Returns the secondary Physical Layer of the advertisement PHY_LE_1M = 1, PHY_LE_2M = 2 PHY_LE_CODED = 3, PHY_UNUSED = 0
...		

...		
getTimestampNanos()	3292098953890580	Returns timestamp since boot when the scan record was observed.
getTxPower()	127	Returns the transmit power in dBm [-127,126] TX_POWER_NOT_PRESENT = 127
hashCode()	1656043611	Returns a hashcode value for the current object
isConnectable()	true	Returns boolean if this object represents a connectable scanr esult
isLegacy()	true	Returns true if this object represents a legacy scan result. Those do not contain advanced advertising information as specified in Bluetooth Core Specification v5.
toString()	ScanResult{ device=DD:DB:24:A3:1B:54, scanRecord= ScanRecord [mAdvertiseFlags=-1, mServiceUuids=null, mServiceSolicitationUuids=[], mManufacturerSpecificData={76 =[18, 25, 32, 78, 115, 40, 1, 55, 11, 9, -74, -9, 78, 70, 53, -100, -111, 39, 17, 13, 28, 82, 91, 71, -40, 2, 84]}, mServiceData={}, mTxPowerLevel=-2147483648, mDeviceName=null, mTDSData=null], rssi=-50, timestampNanos=3292098953890580, eventType=27, primaryPhy=1, secondaryPhy=0, advertisingSid=255, txPower=127, periodicAdvertisingInterval=0}	String Representation of object
The description of the functions is based on: https://developer.android.com/reference/android/bluetooth/le/ScanResult Not included Functions and why: • equals(): comparing ScanResults while analyzing ScanResults was neither possible nor necessary • writeToParce(): this function included no property or constant containing valuable information		

Figure 37: Appendix: ScanResult of a Chipolo ONE Spot.

Example ScanRecord of a Chipolo ONE Spot		
Method	**Value**	**Description**
getAdvertiseFlags()	-1	Returns the advertising flags indicating the discoverable mode and capability of the device. Returns -1 if the flag field is not set.
getBytes()	[B@1c8431b	Returns raw bytes of scan record.
getDeviceName()	null	Returns the local name of the BLE device.
getManufacturerSpecificData()	{76=[B@898fcb8}	Returns a sparse array of manufacturer identifier and its corresponding manufacturer specific data.
getServiceData()	{}	Returns the service data byte array associated with the serviceUuid
getServiceSolicitationUuids()	[]	Returns a list of service solicitation UUIDs within the advertisement that are used to identify the Bluetooth GATT services.
getServiceUuids()	null	Returns a list of service UUIDs within the advertisement that are used to identify the bluetooth GATT services.
...		

...		
getTxPowerLevel()	-2147483648	Returns the transmission power level of the packet in dBm. Returns -2147483648 if the field is not set. This value can be used to calculate the path loss of a received packet using the following equation: pathloss = txPowerLevel - rssi
toString()	ScanRecord [mAdvertiseFlags=-1, mServiceUuids=null, mServiceSolicitationUuids=[], mManufacturerSpecificData={76 =[18, 25, 32, 78, 115, 40, 1, 55, 11, 9, -74, -9, 78, 70, 53, -100, -111, 39, 17, 13, 28, 82, 91, 71, -40, 2, 84]}, mServiceData={}, mTxPowerLevel=-2147483648, mDeviceName=null, mTDSData=null]	String Representation of object
The description of the functions is based on: https://developer.android.com/reference/android/bluetooth/le/ScanRecord Not included Functions and why: • getAdvertisingDataMap(): has been added in API level 33		

Figure 38: Appendix: ScanRecord of a Chipolo ONE Spot.

Example BluetoothDevice of a Chipolo ONE Spot		
Method	**Value**	**Description**
fetchUuidsWithSdp()	true	Perform a service discovery on the remote device to get the UUIDs supported. Returns a boolean.
getAddress()	DD:DB:24:A3:1B:54	Returns the hardware address of this BluetoothDevice.
getAlias()	null	Get the locally modifiable name (alias) of the remote Bluetooth device.
getBluetoothClass()	0	return Bluetooth Class of Remote device PROFILE_HEADSET = 0
getBondState()	10	Get the bond state of the remote device. BONDE_NONE = 10 BOND_BONDING = 11 BOND_BONDED = 12
getName()	null	Get the friendly Bluetooth name of the remote device.
...		

...		
getType()	0	Get the Bluetooth device type of the remote device. DEVICE_TYPE_CLASSIC = 1 DEVICE_TYPE_LE = 2 DEVICE_TYPE_DUAL = 3 DEVICE_TYPE_UNKNOWN = 0
getUuids()	null	Returns the supported features (UUIDs) of the remote device.
toString()	DD:DB:24:A3:1B:54	string representation of object
The description of the functions is based on: https://developer.android.com/reference/android/bluetooth/BluetoothDevice Not included Functions and why: • connectGatt() up to equals(): no connection was in scope as well comparing to other device • hashCode(): creates a hash of the object which for this task is of no further use • setAliats() up to setPin(): setters are of no help analyzing a bluetooth device		

Figure 39: Appendix: BluetoothDevice of a Chipolo ONE Spot.

A.3 Galaxy SmartTag+

Example ScanResult of a Galaxy SmartTag+		
Method	**Value**	**Description**
describeContents()	0	Describe the kinds of special objects contained in this Parcelable instance's marshaled representation.
getAdvertisingSid()	255	Returns the advertsing id SID_NOT_PRESENT = 255
getDataStauts()	0	Returns the data Status DATA_COMPLETE = 0 DATA_TRUNCATED = 2
getDevice()	see: Example BluetoothDevice for a Galaxy SmartTag+	Returns remote BluetoothDevice
getPeriodicAdvertisingInterval()	0	Returns the periodic advertising interval in units of 1.25ms between 6 (7.5ms) and 65536 (81918.75ms) PERIODIC_INTERVAL_NOT_PRESENT = 0
getPrimaryPhy()	1	Returns the primary Physical Layer of the advertisement PHY_LE_1M = 1 PHY_LE_CODED = 3
getRssi()	-34	Returns the received signal strength in dBm within the range of [-127, 126]
getScanRecord()	see: Example ScanRecord for a Galaxy SmartTag+	Returns an instance of a ScanRecord
getSecondaryPhy()	0	Returns the secondary Physical Layer of the advertisement PHY_LE_1M = 1, PHY_LE_2M = 2 PHY_LE_CODED = 3, PHY_UNUSED = 0
...		

...		
getTimestampNanos()	3297626644218419	Returns timestamp since boot when the scan record was observed.
getTxPower()	127	Returns the transmit power in dBm [-127,126] TX_POWER_NOT_PRESENT = 127
hashCode()	-1091512122	Returns a hashcode value for the current object
isConnectable()	true	Returns boolean if this object represents a connectable scanr esult
isLegacy()	true	Returns true if this object represents a legacy scan result. Those do not contain advanced advertising information as specified in Bluetooth Core Specification v5.
toString()	ScanResult{ device=6A:9F:CA:2C:F7:87, scanRecord=ScanRecord [mAdvertiseFlags=4, mServiceUuids= [0000fd5a-0000-1000-8000-00805f9b34fb], mServiceSolicitationUuids=[], mManufacturerSpecificData={}, mServiceData={0000fd5a-0000-1000-8000-00805f9b34fb=[19, -20, 56, 1, 111, -84, 74, 90, -36, -115, -36, -14, -74, 0, 0, 0, -71, -54, -82, 111]}, mTxPowerLevel=-2147483648, mDeviceName=Smart Tag, mTDSData=null], rssi=-34, timestampNanos=3297626644218419, eventType=27, primaryPhy=1, secondaryPhy=0, advertisingSid=255, txPower=127, periodicAdvertisingInterval=0}	String Representation of object
The description of the functions is based on: https://developer.android.com/reference/android/bluetooth/le/ScanResult Not included Functions and why: • equals(): comparing ScanResults while analyzing ScanResults was neither possible nor necessary • writeToParce(): this function included no property or constant containing valuable information		

Figure 40: Appendix: ScanResult of a Galaxy SmartTag+.

Example ScanRecord of a Galaxy SmartTag+		
Method	**Value**	**Description**
getAdvertiseFlags()	4	Returns the advertising flags indicating the discoverable mode and capability of the device. Returns -1 if the flag field is not set.
getBytes()	[B@8018866	Returns raw bytes of scan record.
getDeviceName()	Smart Tag	Returns the local name of the BLE device.
getManufacturerSpecificData()	{}	Returns a sparse array of manufacturer identifier and its corresponding manufacturer specific data.
getServiceData()	{0000fd5a-0000-1000-8000-00805f9b34fb=[19, -20, 56, 1, 111, -84, 74, 90, -36, -115, -36, -14, -74, 0, 0, 0, -71, -54, -82, 111]}	Returns the service data byte array associated with the serviceUuid
getServiceSolicitationUuids()	[]	Returns a list of service solicitation UUIDs within the advertisement that are used to identify the Bluetooth GATT services.
getServiceUuids()	[0000fd5a-0000-1000-8000-00805f9b34fb]	Returns a list of service UUIDs within the advertisement that are used to identify the bluetooth GATT services.
...		

...		
getTxPowerLevel()	-2147483648	Returns the transmission power level of the packet in dBm. Returns -2147483648 if the field is not set. This value can be used to calculate the path loss of a received packet using the following equation: pathloss = txPowerLevel - rssi
toString()	ScanRecord [mAdvertiseFlags=4, mServiceUuids= [0000fd5a-0000-1000-8000-00805f9b34fb], mServiceSolicitationUuids=[], mManufacturerSpecificData={}, mServiceData={0000fd5a-0000-1000-8000-00805f9b34fb=[19, -20, 56, 1, 111, -84, 74, 90, -36, -115, -36, -14, -74, 0, 0, 0, -71, -54, -82, 111]}, mTxPowerLevel=-2147483648, mDeviceName=Smart Tag, mTDSData=null]	String Representation of object
The description of the functions is based on: https://developer.android.com/reference/android/bluetooth/le/ScanRecord Not included Functions and why: • getAdvertisingDataMap(): has been added in API level 33		

Figure 41: Appendix: ScanRecord of a Galaxy SmartTag+.

Example BluetoothDevice of a Galaxy SmartTag+		
Method	**Value**	**Description**
fetchUuidsWithSdp()	true	Perform a service discovery on the remote device to get the UUIDs supported. Returns a boolean.
getAddress()	6A:9F:CA:2C:F7:87	Returns the hardware address of this BluetoothDevice.
getAlias()	Smart Tag	Get the locally modifiable name (alias) of the remote Bluetooth device.
getBluetoothClass()	7936	return Bluetooth Class of Remote device PROFILE_HEADSET = 0 (7936 not declared)
getBondState()	10	Get the bond state of the remote device. BONDE_NONE = 10 BOND_BONDING = 11 BOND_BONDED = 12
getName()	Smart Tag	Get the friendly Bluetooth name of the remote device.
...		

...		
getType()	2	Get the Bluetooth device type of the remote device. DEVICE_TYPE_CLASSIC = 1 DEVICE_TYPE_LE = 2 DEVICE_TYPE_DUAL = 3 DEVICE_TYPE_UNKNOWN = 0
getUuids()	null	Returns the supported features (UUIDs) of the remote device.
toString()	6A:9F:CA:2C:F7:87	string representation of object
The description of the functions is based on: https://developer.android.com/reference/android/bluetooth/BluetoothDevice Not included Functions and why: • connectGatt() up to equals(): no connection was in scope as well comparing to other device • hashCode(): creates a hash of the object which for this task is of no further use • setAliats() up to setPin(): setters are of no help analyzing a bluetooth device		

Figure 42: Appendix: BluetoothDevice of a Galaxy SmartTag+.

A.4 Tile

Example ScanResult of a Tile		
Method	**Value**	**Description**
describeContents()	0	Describe the kinds of special objects contained in this Parcelable instance's marshaled representation.
getAdvertisingSid()	255	Returns the advertsing id SID_NOT_PRESENT = 255
getDataStauts()	0	Returns the data Status DATA_COMPLETE = 0 DATA_TRUNCATED = 2
getDevice()	see: Example BluetoothDevice for a Tile	Returns remote BluetoothDevice
getPeriodicAdvertisingInterval()	0	Returns the periodic advertising interval in units of 1.25ms between 6 (7.5ms) and 65536 (81918.75ms) PERIODIC_INTERVAL_NOT_PRESENT = 0
getPrimaryPhy()	1	Returns the primary Physical Layer of the advertisement PHY_LE_1M = 1 PHY_LE_CODED = 3
getRssi()	-43	Returns the received signal strength in dBm within the range of [-127, 126]
getScanRecord()	see: Example ScanRecord for a Tile	Returns an instance of a ScanRecord
getSecondaryPhy()	0	Returns the secondary Physical Layer of the advertisement PHY_LE_1M = 1, PHY_LE_2M = 2 PHY_LE_CODED = 3, PHY_UNUSED = 0
...		

...		
getTimestampNanos()	3296299765530748	Returns timestamp since boot when the scan record was observed.
getTxPower()	127	Returns the transmit power in dBm [-127,126] TX_POWER_NOT_PRESENT = 127
hashCode()	953065192	Returns a hashcode value for the current object
isConnectable()	true	Returns boolean if this object represents a connectable scanr esult
isLegacy()	true	Returns true if this object represents a legacy scan result. Those do not contain advanced advertising information as specified in Bluetooth Core Specification v5.
toString()	ScanResult {device=DB:EE:05:2C:9F:97, scanRecord=ScanRecord [mAdvertiseFlags=6, mServiceUuids= [0000feed-0000-1000-8000-00805f9b34fb], mServiceSolicitationUuids=[], mManufacturerSpecificData={}, mServiceData={0000feed-0000-1000-8000-00805f9b34fb=[2, 0, 96, 30, 106, -113, -15, -1, -113, -116]}, mTxPowerLevel=-2147483648, mDeviceName=null, mTDSData=null], rssi=-43, timestampNanos=3296299765530748, eventType=27, primaryPhy=1, secondaryPhy=0, advertisingSid=255, txPower=127, periodicAdvertisingInterval=0}	String Representation of object
The description of the functions is based on: https://developer.android.com/reference/android/bluetooth/le/ScanResult Not included Functions and why: • equals(): comparing ScanResults while analyzing ScanResults was neither possible nor necessary • writeToParce(): this function included no property or constant containing valuable information		

Figure 43: Appendix: ScanResult of a Tile.

Example ScanRecord of a Tile		
Method	**Value**	**Description**
getAdvertiseFlags()	6	Returns the advertising flags indicating the discoverable mode and capability of the device. Returns -1 if the flag field is not set.
getBytes()	[B@ca87e3d	Returns raw bytes of scan record.
getDeviceName()	null	Returns the local name of the BLE device.
getManufacturerSpecificData()	{}	Returns a sparse array of manufacturer identifier and its corresponding manufacturer specific data.
getServiceData()	{0000feed-0000-1000-8000-00805f9b34fb=[2, 0, 96, 30, 106, -113, -15, -1, -113, -116]}	Returns the service data byte array associated with the serviceUuid
getServiceSolicitationUuids()	[]	Returns a list of service solicitation UUIDs within the advertisement that are used to identify the Bluetooth GATT services.
getServiceUuids()	[0000feed-0000-1000-8000-00805f9b34fb]	Returns a list of service UUIDs within the advertisement that are used to identify the bluetooth GATT services.
...		

...		
getTxPowerLevel()	-2147483648	Returns the transmission power level of the packet in dBm. Returns -2147483648 if the field is not set. This value can be used to calculate the path loss of a received packet using the following equation: pathloss = txPowerLevel - rssi
toString()	ScanRecord [mAdvertiseFlags=6, mServiceUuids= [0000feed-0000-1000-8000-00805f9b34fb], mServiceSolicitationUuids=[], mManufacturerSpecificData={}, mServiceData={0000feed-0000-1000-8000-00805f9b34fb=[2, 0, 96, 30, 106, -113, -15, -1, -113, -116]}, mTxPowerLevel=-2147483648, mDeviceName=null, mTDSData=null]	String Representation of object
The description of the functions is based on: https://developer.android.com/reference/android/bluetooth/le/ScanRecord Not included Functions and why: • getAdvertisingDataMap(): has been added in API level 33		

Figure 44: Appendix: ScanRecord of a Tile.

Example BluetoothDevice of a Tile		
Method	**Value**	**Description**
fetchUuidsWithSdp()	true	Perform a service discovery on the remote device to get the UUIDs supported. Returns a boolean.
getAddress()	DB:EE:05:2C:9F:97	Returns the hardware address of this BluetoothDevice.
getAlias()	null	Get the locally modifiable name (alias) of the remote Bluetooth device.
getBluetoothClass()	0	return Bluetooth Class of Remote device PROFILE_HEADSET = 0
getBondState()	10	Get the bond state of the remote device. BONDE_NONE = 10 BOND_BONDING = 11 BOND_BONDED = 12
getName()	null	Get the friendly Bluetooth name of the remote device.
...		

...		
getType()	0	Get the Bluetooth device type of the remote device. DEVICE_TYPE_CLASSIC = 1 DEVICE_TYPE_LE = 2 DEVICE_TYPE_DUAL = 3 DEVICE_TYPE_UNKNOWN = 0
getUuids()	null	Returns the supported features (UUIDs) of the remote device.
toString()	DB:EE:05:2C:9F:97	string representation of object
The description of the functions is based on: https://developer.android.com/reference/android/bluetooth/BluetoothDevice Not included Functions and why: • connectGatt() up to equals(): no connection was in scope as well comparing to other device • hashCode(): creates a hash of the object which for this task is of no further use • setAliats() up to setPin(): setters are of no help analyzing a bluetooth device		

Figure 45: Appendix: BluetoothDevice of a Tile.

Appendix B

Experiments

B.1 RSSI

All measurements of the RSSI experiments can be found here: `https://github.com/LouisBienz/HomeScout/tree/main/experiements/RSSI`

B.2 Tracking Algorithm

All measurements from every run testing the tracking algorithm can be found here: `https://github.com/LouisBienz/HomeScout/tree/main/experiements/Tracking_Algorithm`

B.3 Long Time Unpaired and Alternating States

B.3.1 AirTag

Measured Results for AirTag

LONG TIME UNPAIRED

```
State transition: Paired --> Unpaired

t = 0
AirTag with MAC address = F0:E1:35:8E:39:0C

t = 5
AirTag with MAC address = F0:E1:35:8E:39:0C

t = 10
AirTag with MAC address = F0:E1:35:8E:39:0C

t = 15
AirTag with MAC address = F0:E1:35:8E:39:0C

t = 20
AirTag with MAC address = F0:E1:35:8E:39:0C

t = 25
AirTag with MAC address = F0:E1:35:8E:39:0C

t = 30
AirTag with MAC address = F0:E1:35:8E:39:0C

t = 35
AirTag with MAC address = F0:E1:35:8E:39:0C

t = 40
AirTag with MAC address = F0:E1:35:8E:39:0C

t = 45
AirTag with MAC address = F0:E1:35:8E:39:0C

t = 50
AirTag with MAC address = F0:E1:35:8E:39:0C

t = 55
AirTag with MAC address = F0:E1:35:8E:39:0C

t = 60
AirTag with MAC address = F0:E1:35:8E:39:0C

State transition: Unpaired --> Paired
```

ALTERNATING PAIRING STATE

```
t = 0
State transition: Paired --> Unpaired
AirTag with MAC address = FD:75:4F:02:FA:AB
State transition: Unpaired --> Paired

t = 5
State transition: Paired --> Unpaired
AirTag with MAC address = FD:75:4F:02:FA:AB
State transition: Unpaired --> Paired

t = 10
State transition: Paired --> Unpaired
AirTag with MAC address = FD:75:4F:02:FA:AB
State transition: Unpaired --> Paired

t = 15
State transition: Paired --> Unpaired
AirTag with MAC address = D7:21:4F:A4:9B:E8
State transition: Unpaired --> Paired

t = 20
State transition: Paired --> Unpaired
AirTag with MAC address = D7:21:4F:A4:9B:E8
State transition: Unpaired --> Paired

t = 25
State transition: Paired --> Unpaired
AirTag with MAC address = D7:21:4F:A4:9B:E8
State transition: Unpaired --> Paired

t = 30
State transition: Paired --> Unpaired
AirTag with MAC address = C8:C6:C8:7A:74:AF
State transition: Unpaired --> Paired

t = 35
State transition: Paired --> Unpaired
AirTag with MAC address = C8:C6:C8:7A:74:AF
State transition: Unpaired --> Paired

t = 40
State transition: Paired --> Unpaired
AirTag with MAC address = C8:C6:C8:7A:74:AF
State transition: Unpaired --> Paired

t = 45
State transition: Paired --> Unpaired
AirTag with MAC address = D8:35:92:36:29:7E
State transition: Unpaired --> Paired


t = 50
State transition: Paired --> Unpaired
AirTag with MAC address = D8:35:92:36:29:7E
State transition: Unpaired --> Paired

t = 55
State transition: Paired --> Unpaired
AirTag with MAC address = D8:35:92:36:29:7E
State transition: Unpaired --> Paired

t = 60
State transition: Paired --> Unpaired
AirTag with MAC address = CD:FE:8D:CE:E1:A9
State transition: Unpaired --> Paired
```

Figure 46: Appendix: Measured MAC addresses of an AirTag for the long time unpaired and alternating states experiments.

B.3.2 Chipolo ONE Spot

Measured Results for Chipolo One Spot

LONG TIME UNPAIRED

```
State transition: Paired --> Unpaired

t = 0
Chipolo with MAC address = D0:B6:68:CE:96:C2

t = 5
Chipolo with MAC address = D0:B6:68:CE:96:C2

t = 10
Chipolo with MAC address = D0:B6:68:CE:96:C2

t = 15
Chipolo with MAC address = D0:B6:68:CE:96:C2

t = 20
Chipolo with MAC address = D0:B6:68:CE:96:C2

t = 25
Chipolo with MAC address = D0:B6:68:CE:96:C2

t = 30
Chipolo with MAC address = D0:B6:68:CE:96:C2

t = 35
Chipolo with MAC address = D0:B6:68:CE:96:C2

t = 40
Chipolo with MAC address = D0:B6:68:CE:96:C2

t = 45
Chipolo with MAC address = D0:B6:68:CE:96:C2

t = 50
Chipolo with MAC address = D0:B6:68:CE:96:C2

t = 55
Chipolo with MAC address = D0:B6:68:CE:96:C2

t = 60
Chipolo with MAC address = D0:B6:68:CE:96:C2

State transition: Unpaired --> Paired
```

ALTERNATING PAIRING STATE

```
t = 0
State transition: Paired --> Unpaired
Chipolo with MAC address = DF:3D:4A:3A:C5:31
State transition: Unpaired --> Paired

t = 5
State transition: Paired --> Unpaired
Chipolo with MAC address = DF:3D:4A:3A:C5:31
State transition: Unpaired --> Paired

t = 10
State transition: Paired --> Unpaired
Chipolo with MAC address = DF:3D:4A:3A:C5:31
State transition: Unpaired --> Paired

t = 15
State transition: Paired --> Unpaired
Chipolo with MAC address = E2:CA:05:1C:3C:88
State transition: Unpaired --> Paired

t = 20
State transition: Paired --> Unpaired
Chipolo with MAC address = E2:CA:05:1C:3C:88
State transition: Unpaired --> Paired

t = 25
State transition: Paired --> Unpaired
Chipolo with MAC address = E2:CA:05:1C:3C:88
State transition: Unpaired --> Paired

t = 30
State transition: Paired --> Unpaired
Chipolo with MAC address = D3:41:26:41:4A:F7
State transition: Unpaired --> Paired

t = 35
State transition: Paired --> Unpaired
Chipolo with MAC address = D3:41:26:41:4A:F7
State transition: Unpaired --> Paired

t = 40
State transition: Paired --> Unpaired
Chipolo with MAC address = D3:41:26:41:4A:F7
State transition: Unpaired --> Paired

t = 45
State transition: Paired --> Unpaired
Chipolo with MAC address = F5:BD:D6:6E:34:49
State transition: Unpaired --> Paired


t = 50
State transition: Paired --> Unpaired
Chipolo with MAC address = F5:BD:D6:6E:34:49
State transition: Unpaired --> Paired

t = 55
State transition: Paired --> Unpaired
Chipolo with MAC address = F5:BD:D6:6E:34:49
State transition: Unpaired --> Paired

t = 60
State transition: Paired --> Unpaired
Chipolo with MAC address = D7:8F:00:08:DA:AA
State transition: Unpaired --> Paired
```

Figure 47: Appendix: Measured MAC addresses of a Chipolo ONE Spot for the long time unpaired and alternating states experiments.

B.3.3 Galaxy SmartTag+

Measured Results for Galaxy SmartTag+

LONG TIME UNPAIRED

State transition: Paired --> Unpaired

t = 0
SmartTag with MAC address = 6C:8D:C3:87:33:64

t = 5
SmartTag with MAC address = 54:7A:4E:5D:92:4F

t = 10
SmartTag with MAC address = 54:7A:4E:5D:92:4F

t = 15
SmartTag with MAC address = 55:5B:09:61:64:3A

t = 20
SmartTag with MAC address = 43:EC:92:C6:87:F2

t = 25
SmartTag with MAC address = 43:EC:92:C6:87:F2

t = 30
SmartTag with MAC address = 6F:C0:7B:B8:AF:85

t = 35
SmartTag with MAC address = 63:0F:8B:9E:8F:48

t = 40
SmartTag with MAC address = 63:0F:8B:9E:8F:48

t = 45
SmartTag with MAC address = 6A:FF:9B:16:34:57

t = 50
SmartTag with MAC address = 78:AE:BB:E6:71:79

t = 55
SmartTag with MAC address = 78:AE:BB:E6:71:79

t = 60
SmartTag with MAC address = 5B:50:A6:4B:B3:EA

State transition: Unpaired --> Paired

ALTERNATING PAIRING STATE

t = 0
State transition: Paired --> Unpaired
SmartTag with MAC address = 6E:E6:1B:90:F8:E1
State transition: Unpaired --> Paired

t = 5
State transition: Paired --> Unpaired
SmartTag with MAC address = 53:F6:4C:1A:F7:2C
State transition: Unpaired --> Paired

t = 10
State transition: Paired --> Unpaired
SmartTag with MAC address = 7E:E9:26:D7:7A:7B
State transition: Unpaired --> Paired

t = 15
State transition: Paired --> Unpaired
SmartTag with MAC address = 48:47:D7:92:52:E4
State transition: Unpaired --> Paired

t = 20
State transition: Paired --> Unpaired
SmartTag with MAC address = 44:C2:8F:02:3F:75
State transition: Unpaired --> Paired

t = 25
State transition: Paired --> Unpaired
SmartTag with MAC address = 7B:25:84:7F:0A:13
State transition: Unpaired --> Paired

t = 30
State transition: Paired --> Unpaired
SmartTag with MAC address = 5C:3C:EF:76:8B:B1
State transition: Unpaired --> Paired

t = 35
State transition: Paired --> Unpaired
SmartTag with MAC address = 7D:07:06:D2:56:2A
State transition: Unpaired --> Paired

t = 40
State transition: Paired --> Unpaired
SmartTag with MAC address = 53:83:EF:0A:EA:B4
State transition: Unpaired --> Paired

t = 45
State transition: Paired --> Unpaired
SmartTag with MAC address = 60:6A:1E:81:4C:D7
State transition: Unpaired --> Paired

t = 50
State transition: Paired --> Unpaired
SmartTag with MAC address = 4B:A1:15:A7:74:35
State transition: Unpaired --> Paired

t = 55
State transition: Paired --> Unpaired
SmartTag with MAC address = 57:E7:C0:FD:7D:24
State transition: Unpaired --> Paired

t = 60
State transition: Paired --> Unpaired
SmartTag with MAC address = 64:AF:19:F9:E1:02
State transition: Unpaired --> Paired

Figure 48: Appendix: Measured MAC addresses of a Galaxy SmartTag+ for the long time unpaired and alternating states experiments.

B.3.4 Tile

Measured Results for Tile

LONG TIME UNPAIRED

```
State transition: Paired --> Unpaired

t = 0
Tile with MAC address = DB:EE:05:2C:9F:97

t = 5
Tile with MAC address = DB:EE:05:2C:9F:97

t = 10
Tile with MAC address = DB:EE:05:2C:9F:97

t = 15
Tile with MAC address = DB:EE:05:2C:9F:97

t = 20
Tile with MAC address = DB:EE:05:2C:9F:97

t = 25
Tile with MAC address = DB:EE:05:2C:9F:97

t = 30
Tile with MAC address = DB:EE:05:2C:9F:97

t = 35
Tile with MAC address = DB:EE:05:2C:9F:97

t = 40
Tile with MAC address = DB:EE:05:2C:9F:97

t = 45
Tile with MAC address = DB:EE:05:2C:9F:97

t = 50
Tile with MAC address = DB:EE:05:2C:9F:97

t = 55
Tile with MAC address = DB:EE:05:2C:9F:97

t = 60
Tile with MAC address = DB:EE:05:2C:9F:97

State transition: Unpaired --> Paired
```

ALTERNATING PAIRING STATE

```
t = 0
State transition: Paired --> Unpaired
Tile with MAC address = DB:EE:05:2C:9F:97
State transition: Unpaired --> Paired

t = 5
State transition: Paired --> Unpaired
Tile with MAC address = DB:EE:05:2C:9F:97
State transition: Unpaired --> Paired

t = 10
State transition: Paired --> Unpaired
Tile with MAC address = DB:EE:05:2C:9F:97
State transition: Unpaired --> Paired

t = 15
State transition: Paired --> Unpaired
Tile with MAC address = DB:EE:05:2C:9F:97
State transition: Unpaired --> Paired

t = 20
State transition: Paired --> Unpaired
Tile with MAC address = DB:EE:05:2C:9F:97
State transition: Unpaired --> Paired

t = 25
State transition: Paired --> Unpaired
Tile with MAC address = DB:EE:05:2C:9F:97
State transition: Unpaired --> Paired

t = 30
State transition: Paired --> Unpaired
Tile with MAC address = DB:EE:05:2C:9F:97
State transition: Unpaired --> Paired

t = 35
State transition: Paired --> Unpaired
Tile with MAC address = DB:EE:05:2C:9F:97
State transition: Unpaired --> Paired

t = 40
State transition: Paired --> Unpaired
Tile with MAC address = DB:EE:05:2C:9F:97
State transition: Unpaired --> Paired

t = 45
State transition: Paired --> Unpaired
Tile with MAC address = DB:EE:05:2C:9F:97
State transition: Unpaired --> Paired

t = 50
State transition: Paired --> Unpaired
Tile with MAC address = DB:EE:05:2C:9F:97
State transition: Unpaired --> Paired

t = 55
State transition: Paired --> Unpaired
Tile with MAC address = DB:EE:05:2C:9F:97
State transition: Unpaired --> Paired

t = 60
State transition: Paired --> Unpaired
Tile with MAC address = DB:EE:05:2C:9F:97
State transition: Unpaired --> Paired
```

Figure 49: Appendix: Measured MAC addresses of a Tile for the long time unpaired and alternating states experiments.