INTERNSHIP REPORT – Communication Systems Group, Prof. Dr. Burkhard Stiller

Detection of Bluetooth Low Energy Trackers

Alexandre DOYEN
Lannion, France

Supervisors: Katharina O.E. Müller; Bruno Rodrigez
Date of Submission: August 26, 2022

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

ifi

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

Acknowledgments

To begin with, thank you to my supervisor, Katharina O.E. Müller, and the head of the CSG team, Dr. Prof. Burkhard Stiller, without whom this internship not have taken place. Therefore, thank you to Laurent d'Orazio who proposed me this internship subject and helped me to find this.

Then, thank you very much to the CSG team for their welcome at Zürich. We have passed a lot of funny moments, and you have helped me to be integrated into the team!

Contents

Acknowledgments **i**

1 Introduction **1**

1.1 Motivation 1

1.2 Description of Work 1

2 Understanding Bluetooth Low Energy protocol **3**

2.1 Goal 3

2.2 Capturing and analyzing packets 3

2.2.1 Capturing packets 3

2.2.2 Cleaning captures files 4

2.3 Understanding BLE-LL packets 5

2.3.1 Different type of PDUs 5

2.3.2 Different kind of PDU payloads for advertising packets 6

3 Analysis about Apple AirTags **9**

3.1 Generalities about Apple AirTags 9

3.1.1 Operation for an average user 9

3.1.2 Properties 10

3.2 AirGuard 10

3.3 Methodology 11

3.4 Observations 11

3.4.1 When Apple AirTag is unpaired 11

3.4.2 When is paired and closest to the enabled iPhone 13
3.5 Summary 13

4 Analysis about Tiles 15
4.1 Generalities about Tiles 15
4.2 Technical aspects 16
4.3 Detection of Tile tags 16

5 OpenHaystack 17
5.1 Introduction 17
5.2 Setup of the experimentation 17
5.2.1 On the computer 17
5.2.2 The fake tag's setup 18
5.3 Experimentation 19
5.4 Summary 21

6 Chipolo One Spot 23
6.1 Introduction 23
6.2 Strategy to detect it in the air 23
6.3 Experimentation to fake it with the Raspberry Pi 24
6.4 Generalization 26
6.5 Summary 26

7 Samsung Galaxy SmartTag+ 27
7.1 Introduction 27
7.2 Analysis 28
7.3 Summary 29

8 Summary and Conclusions **31**

8.1 Summary 31

8.2 How to detect each tags ? 31

8.2.1 Apple AirTag 31

8.2.2 Tiles 32

8.2.3 OpenHaystack and Chipolo One Spot 32

8.2.4 Samsung Galaxy SmartTag+ 32

8.3 Conclusion 33

Bibliography **A**

Glossary **C**

Acronyms **E**

List of Figures **H**

A Time diagrams of packet transmitted by when unpaired and iPhone on **I**

B Generation of the picture of OpenHaystack reports **K**

Chapter 1

Introduction

1.1 Motivation

The topic was about the detection of BLE tackers around a user, such as Apple AirTags. In fact, these trackers are very tiny and could be hidden in persons' bags to track them. So, it explains why we need to have means to detect trackers around us and to be warned if there are suspect trackers.

1.2 Description of Work

To do that, the first thing was to understand the BLE protocol, and how devices communicate with each other. The final goal is to develop a Flutter application to let users check if there are trackers around them.

Here is the list of trackers used for the project :

- Apple AirTag
- Samsung Galaxy SmartTag+
- Chipolo One Spot
- Tile

Chapter 2

Understanding Bluetooth Low Energy protocol

2.1 Goal

The main goal of BLE is a specification of a wireless protocol to enable devices to communicate with each other with less energy than the ”standard Bluetooth”. It is important to take note BLE is different from Bluetooth.

The key point is IOTs systems have to send only small data (Up to 256 bytes for each packet [4]). So, to send these packets, the consumption of energy is reduced. Then, for example, Apple says the battery of an Apple AirTag can work for more than one year[2], and we will see Apple AirTags send a lot of small packets.

2.2 Capturing and analyzing packets

2.2.1 Capturing packets

To capture packets, I have used a Ubertooth One USB key to capture packets. This tool lets me capture BLE packets that are in the area. The figure 2.1 shows that key.

Figure 2.1: Ubertooth One USB key

Then, I have a capture file that contains each packet that the key has captured during the time of the capture. The Wireshark software lets seeing these packets. The time of captures was five minutes because it lets enough time to catch a lot of packet, and have a representation of the type of packets sent in the environment.

To analyze data, there is a useful tool: R. It can be used to produce visualizations to help to find some things and study some characteristics of devices.[1].

2.2.2 Cleaning captures files

When we send data wirelessly, there is a lot of data loss. So, I focus my analysis on well-formed packets. A well-formed packet is a packet where its size is coherent (Size of packet = Size of header + Size of payloads), and its CRC check is good by using a specific algorithm [9].

Here is the filter used in Wireshark to filter these packets :

```
(frame.protocols == "ppi:pcap_pktdata:user_dlt:btle:btcommon")
&& !(_ws.malformed) && !(btle.crc.incorrect)
```

[1]Lot of visualizations of this report have been made by R

This filter permits to only keep well-formed packets, with coherent size and good CRC check.

It is important to take note this filter keeps only from 60 to 80 percent of packets. It will be important to understand the visualizations that will come after in this report.

2.3 Understanding BLE-LL packets

In this section, I explain how is built a BTLE Link-Layer packet with a focus on important information for this project. The figure 2.2 present how is crafted a BLE-LL packet. The first field, which is the preamble, is useless for us.[2] The access address is also useless to us. The PDU field contains very useful information for us because it contains payloads and the advertising address. Finally, the CRC field stands to ensure the received packet is the same as the sent packet.

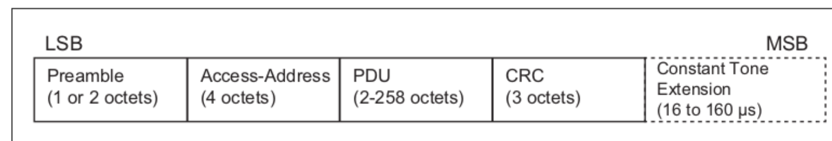

Figure 2.2: Diagram of BLE-LE packet

So, let's know more about PDU. PDU always begins with a header, which contains PDU type, some flags, and height bits for the length of the payload. Figure 2.3 present PDU header structure. Then, after the header, there is the payload.

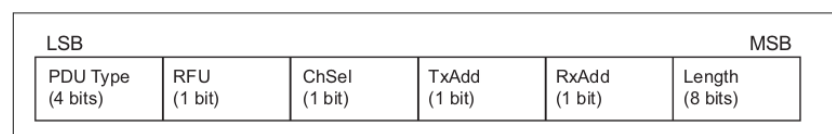

Figure 2.3: Diagram of PDU header fields

2.3.1 Different type of PDUs

There is some type of BLE packets, and interesting types for us are :

1. `ADV_IND`
2. `ADV_NONCONN_IND`
3. `ADV_SCAN_IND`

[2]See section 2 of part B of volume 6 of Bluetooth specifications[7]

4. `SCAN_REQ`
5. `SCAN_RSP`

The first one stands for Advertising Indications (*ADV_IND*). It is used where a peripheral requests connections to any devices. The second one stands for Advertising Non-Connectable Indications (*ADV_NONCONN_IND*). It is the same as (*ADV_IND*), but it didn't request any connection. Advertising Scan Indications (*ADV_SCAN_IND*) is similar to (*ADV_NONCONN_IND*), but there are additional information with Scan Responses (*SCAN_RSP*).[6]

For an advertisement PDU, there is an advertising address. It contains the advertiser's public address or a randomly generated address. Figure 2.4 contains a view of PDU payload for an *ADV** packet.

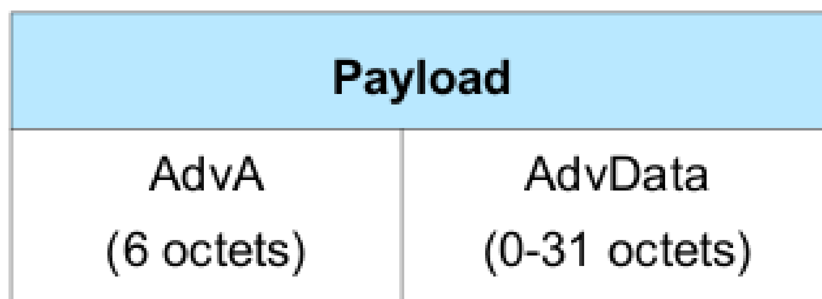

Figure 2.4: Diagram of fields of advertisement PDU payload

The *SCAN_RSP*, which stands for Scan Response is used to answer a Scan Request (*SCAN_REQ*).[7]

2.3.2 Different kind of PDU payloads for advertising packets

In each packets of type `ADV*` (`ADV_IND`, `ADV_NONCONN_IND`, ...), there is a payload. Here it's explained each type of payload met during my experimentation:[11]

- 16 bits service class UUID
- 16 bits UUID service data
- Flags
- Manufacturer-specific data

With each payload, there is a size, coded on one byte. This field contains the total size of the payload, including the size field itself.

It is important to take note each packet does not embed all of these payload types.

16 bits service UUID

First, the class payload describes services that the device offers. Each service is identified by an UUID.

Then, the data payload contains data associated with a service.

Flags

When the advertising packet is connectable, some flags are sent. These flags contain information about the capacities of the device.

Manufacturer specific data

Contains custom data identified by a company identifier (`0x004c` stand for Apple for example). The reference [5] contains a full list of identifiers by companies.

Opening...

There is still another type of payload, like ”Local Name”, to send the name of the device, but it's not used by our tags.

Chapter 3

Analysis about Apple AirTags

3.1 Generalities about Apple AirTags

3.1.1 Operation for an average user

When a user buys an Apple AirTag, he has to pair it with his Apple ID account. Then, the Apple AirTag is registered and entered into the Apple Find My network, and it can be found by its owner. In figure 3.1, there are a view of the Apple AirTag.

To do that, the Apple Find My network is a giant mesh network, where each node is other consumer devices, like iPhone, Mac, etc. Apple AirTag sends packets over the air, to be caught by other Apple devices. Then, these devices send their localization with a message like "Hey! I have seen this Apple AirTag at Strasburgstrasse, in Zürich, at 2022/08/20."[11]

Moreover, it uses the UWB technology to locate it even if the phone which makes the report is far away.

Figure 3.1: An Apple AirTag

3.1.2 Properties

An Apple AirTag has a NFC tag to let a potential finder scan it with a smartphone possessing NFC.

Then, if we scan an unpaired Apple AirTag, we have this address :

```
https://found.apple.com/airtag?pid=5500
        &b=00
        &pt=004c
        &fv=001012d0
        &dg=00
        &z=00
        &bt=df5152442b02
        &sr=HGJGM0UPP0GV
        &bp=0013
```

So, the parameter `bt` is interesting, because it tells us the BLE MAC address of the Apple AirTag : Here, it's `DF:51:52:44:2b:02`.[3]

But, when it's paired, the NFC-provided address changes :

```
https://found.apple.com/airtag?pid=5500
        &b=00
        &pt=004c
        &fv=001012d0
        &dg=00
        &z=00
        &pi=8a9304008c81ab5291a58e8eeaa95a
                28e3206cde99064a1207811b23
```

The pi parameter stands for "Public identity", and is updated every 15 minutes by the Apple AirTag itself.[3] Unfortunately, this value hasn't sense for us, because it's encrypted information.

3.2 AirGuard

AirGuard is a project developed by the university of Darmstadt, in Germany.[8]. It's an open-sourced application that helps users to detect Apple Find My devices around them.

Because this project is open-source, I had been able to check how they find if an Apple device is an Apple AirTag or not, and I have found this formula in the source code :

$$t = (d[2] \& \bar{3}0^{16}) >> 4$$ [1]

[1] & stands for the bitwise binary AND operator, and >> for the binary right shift operator.

Where there are :

- t : Device type number
- d : Manufacturer data payload of the sent packet as an array of bytes[2]. The [2] operator means we take only the third byte of the payload[3]. The payload does not contains the manufacturer identifier in the formula.

Then, if $t = 1$, the device is a paired Apple AirTag with an Apple ID account. So, it's findable with Apple Find My.

So, with this, I am able to filter received packets with Ubertooth to keep only Apple AirTag packets.

3.3 Methodology

To begin with, it is important to explain the methodology I have followed. The goal is to understand what kind of data is sent by Apple AirTags, and how we can be sure the target device is an Apple AirTag.

Thereby, I have made four different captures to spy the behavior of Apple AirTag. So, I have reproduced these situations to spy different behaviors of Apple AirTag :

- Unpaired when the iPhone is off
- Unpaired when the iPhone is on
- Pairing and paired when the iPhone is on
- Paired when the iPhone is off and it's Apple Find My advertisement is disabled[4]

Then, I transformed captures files into CSV files, to process them in R, and create visualizations.

3.4 Observations

3.4.1 When Apple AirTag is unpaired

When the Apple AirTag is unpaired, it sends packets at every $\frac{1}{2}$ seconds. But, at approximately every 50 seconds, the situation seems to be chaotic, with a packet sent almost

[2] `uint8_t` in C or C++

[3] We consider arrays are indexed from 0

[4] In iPhone when we turn it off, we can disable its Apple Find My advertisement. I have done that to ensure the iPhone is completely off.

instantly after the previous one, then the next is sent one second after the previous one. Figure 3.2 shows that.

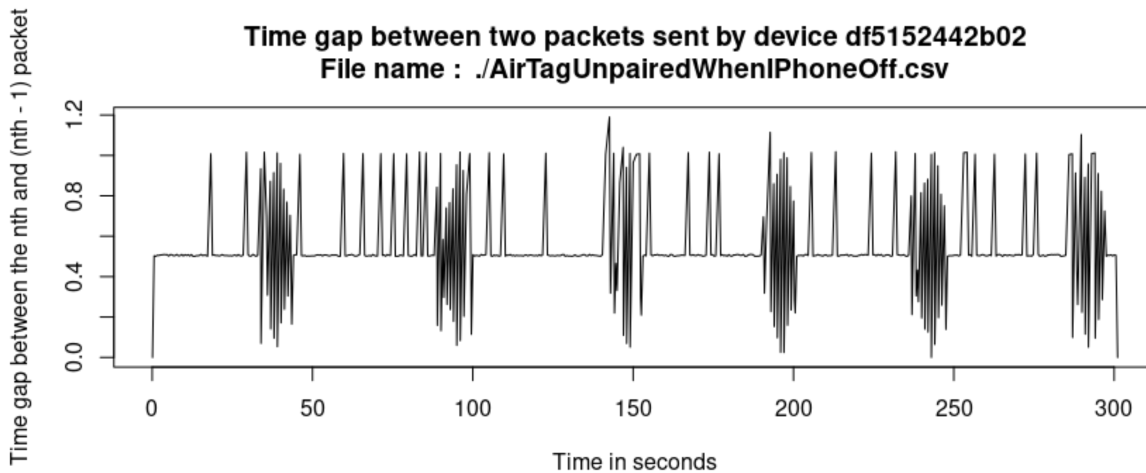

Figure 3.2: Graph showing time gap between two packets when the iPhone is off

Then, it is important to take note as I have written previously, there are a lot of wasted packets, because of CRC errors. So, it's normal to find one peak of 1 second, when around this there are $\frac{1}{2}$ second gap. It just shows at this time (On x axis), there was a malformed packet.

In figure 3.3, we can see the acceleration of the time gap between two packets in this capture.

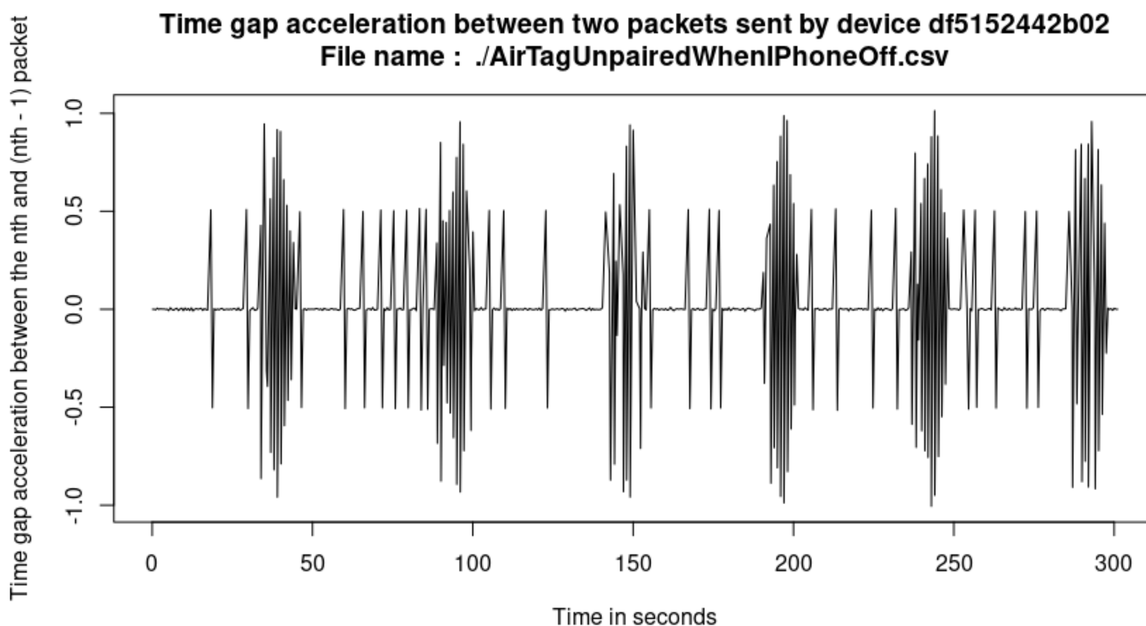

Figure 3.3: Graph showing time gap acceleration between two packets when the iPhone is off

Captures when the iPhone was on are the same, but diagrams are available in appendix A.

3.4.2 When is paired and closest to the enabled iPhone

The main difficulty in this topic is to find my Apple AirTag among others. In fact, the MAC address I have found in the previous section is not usable, because the tag uses a randomly generated MAC address to communicate with iPhone when it's paired to an Apple ID account.

So, I will consider the Apple AirTag[5] that has sent the most amount of packets is good because there is a significative gap between it and the others, as it shown in the table 3.4.

MAC address	Amount of packets sent
4cc6de1514fd	1
600a91297705	7
6e36dfaeadec	37
7447e690f51b	10
e89edf65a795	9
ef07df5f8631	130

Figure 3.4: Table showing number of packet sent during the capture "AirTagPairingAndPairedWhenIPhoneOn.pcapng"

Thus, with this, we can conclude our Apple AirTag's MAC address is `EF:07:DF:5F:86:31`. But, it's important to know this MAC address will change automatically every 15 minutes.

3.5 Summary

Finally, it's useful to only detect paired Apple AirTags, because an unpaired Apple AirTag doesn't send data to the Apple Find My network, and can't be used to track someone or something.

Then, to detect a paired Apple AirTags, it's required to check these points :

- The sent packet only contains one field: "Manufacturer Specific" identified to Apple (`0x004c`)
- The value of $t = (d[2]\&\bar{3}0^{16}) >> 4$ equals 1 where d is the data embed by the "Manufacturer Specific" field of advertising data of packet

[5] Apple AirTags are found with the technique described in 3.2 section

Chapter 4

Analysis about Tiles

4.1 Generalities about Tiles

A tile is a BLE tracker, like Apple AirTag[14]. But, its way of working is different as Apple AirTag. In fact, it uses its own tracking network which is not Apple Find My network. In addition, its BLE packets are not the same as Apple AirTag. For the final user, the real advantage is he doesn't have to have an iPhone or another Apple device to use it. The figure shows an example of a Tile tracker.

Figure 4.1: Example of Tile trackers

Also, there is some partnership with brands, like Intel, to track some things, like laptops[13]. Then, this lets them put Tile chips into devices, to track those with the Tile finding network.

4.2 Technical aspects

All advertising packets sent by tiles are the same: This is `ADV_IND` packets with only these fields in this order :

- Flags
- 16 bits service class UUID
- 16 bits UUID service data

Finally, the packet's size is 60 bytes. In the 16 bits UUID service class field, there are the UUID of Tile[1], which indicates to the device which receives the packet this object carries a Tile service. Then, the 16 bits service data field carries information to track the tag. This information is encrypted, and useless to know for us to test if there is a Tile tag around the phone or not.

4.3 Detection of Tile tags

So, to detect if an advertisement packet is from a Tile, it's only necessary to check if the packet contains the 16 bits service class UUID, and the associated Tile UUID, which is `0xfeed` in big-endian.

[1] `0xfeed`, data are sent in little-endian, so in received packet, it's `0xedfe`.

Chapter 5

OpenHaystack

5.1 Introduction

Like AirGuard, OpenHaystack is a project from the University of Darmstadt, in Germany. Here, the topic is to let users create hand-made tags which can be tracked by the Apple Find My network.[10]

So, users can use different devices to create their tags with these:

- An ESP32 chip
- A device with a nRF51822 chip[1]
- A computer that runs Linux with Python and BLE availability

5.2 Setup of the experimentation

For us, this project could be interesting, because it lets us discover how Apple Find My network works.

5.2.1 On the computer

To do that, I had to install a hackintosh virtual machine on my computer, because this application needs a Mac computer to see reports to the Apple Find My network.

So, to do that, I have used QEMU-KVM to install Mac OS/X on my computer as a virtual machine. Figure 5.1 shows that virtual machine running on my computer.

[1]SOC from Nordic Semiconductors with BLE features.

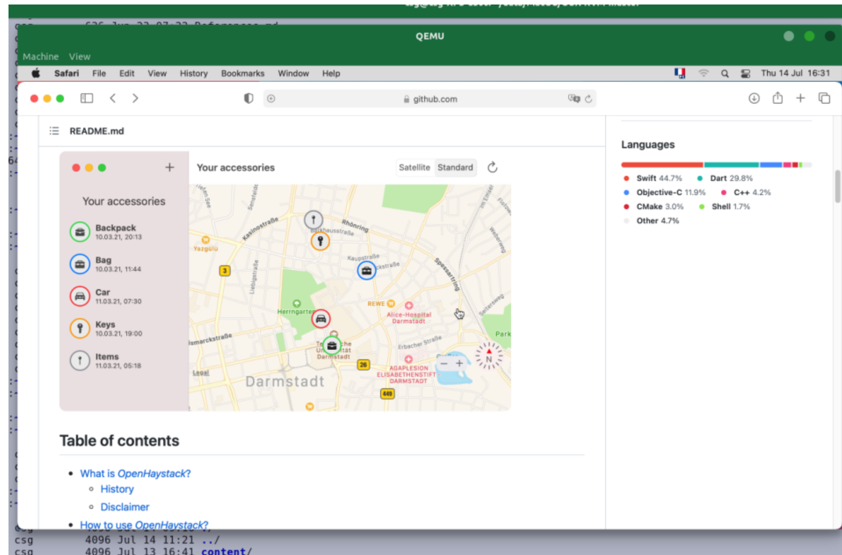

Figure 5.1: Mac OS/X Big Sur (version 11) running on QEMU-KVM

5.2.2 The fake tag's setup

To create the fake tag, it was used a Raspberry Pi, which is a tiny computer that has a size of a credit card (Figure 5.2). It uses an 1GHz single core CPU, 512 MB of RAM, and has BLE.

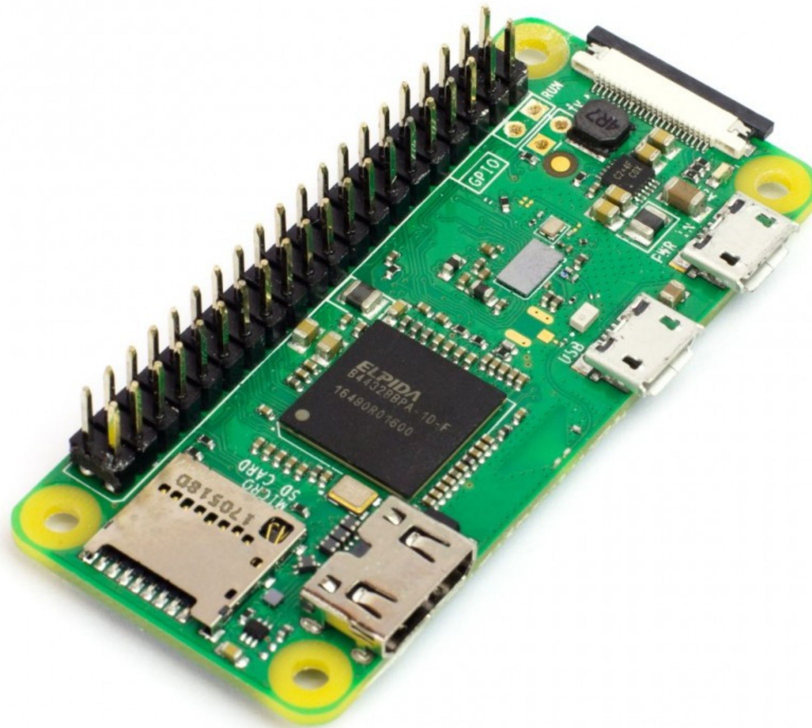

Figure 5.2: Raspberry Pi Zero W, the Raspberry Pi used to test OpenHaystack

On this, there is a Linux operating system, with a Python environment, to run the Python script delivered by the OpenHaystack project to make an Apple Find My findable device.

Then, this script is very interesting, because it tells how works BLE transmissions for non-Apple devices to be findable through the Apple Find My network.

5.3 Experimentation

With all tools ready, it's easy to test OpenHaystack. So, to do that, a device was been generated in the OpenHaystack application, on the hackintosh virtual machine.

Then, the parametrization of the Python script on the Raspberry Pi lets sends advertisement packets for the generated tag. The required parameter is a key, generated by the application on the Mac.

One execution of the script sends one advertisement, so it was required to make a Bash loop to launch repeatedly the Python script. With a promenade in the street with the Raspberry Pi powered by an external battery, and without an iPhone, we can see in figure 5.3 reporting data works fine. The taken route was from the "Psychologisches Institut der Universität Zürich", at the bottom left on the map, to the "Oerlikerhus" tram station, on the top right of the map, and come back to the university.

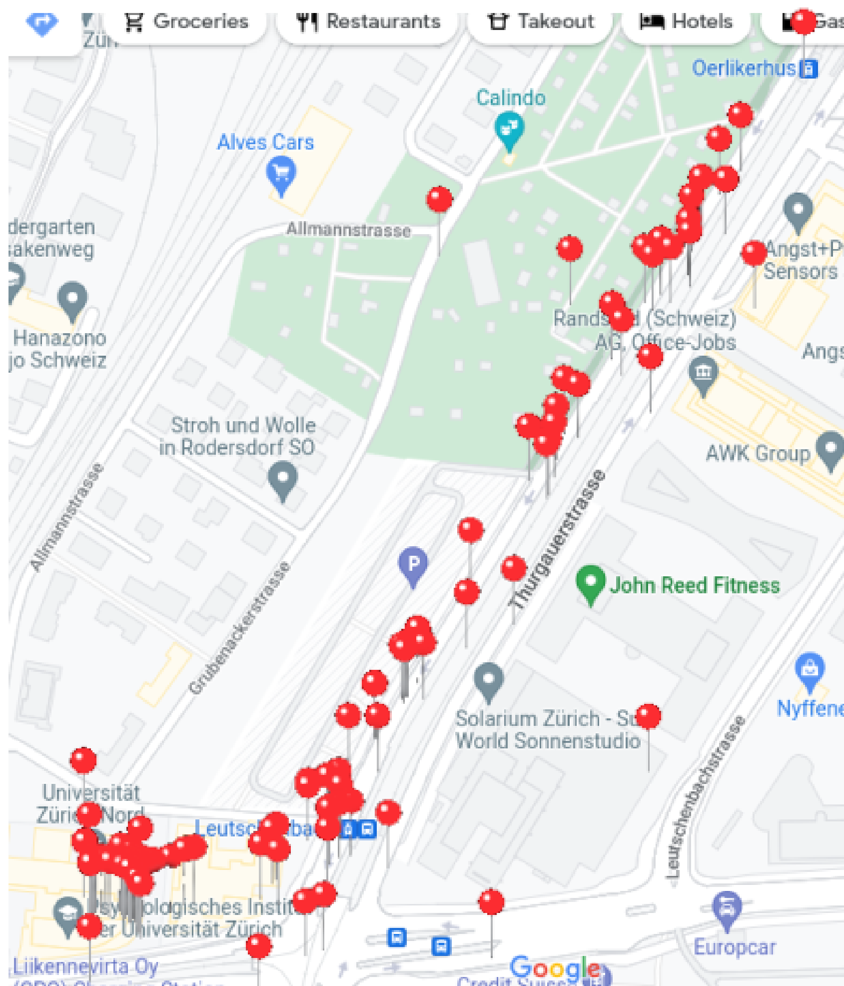

Figure 5.3: Test of OpenHaystack when walking in Oerlikon, at Zürich

Each pin stands for each report to Apple's servers by walkers' iPhones.[2]

5.4 Summary

With these tests and thanks to the University of Darmstadt's work, it's easier to check if a BLE advertising packet which contains a "Manufacturer specific" field with the company ID of Apple is targeted to the Apple Find My tracking network.

However, it's important to remark the OpenHaystack tag is not findable through the "Find My" application on the iPhone, and it's not linked to any Apple ID account. So, someone with some knowledge in computer science can craft a tag to track a victim, without any warning on his phone.

So, according to the Python script, if an Apple's manufacturer-specific advertising data has these characteristics, it's a packet to track the device through the Apple Find My network:[10]

- Its length is 30 (1E in hexadecimal)
- It begins with the sequence "1219"

[2]Appendix B explains how was generated this image.

Chapter 6

Chipolo One Spot

6.1 Introduction

Chipolo one spot is an item tracker made by Chipolo. It's usable only with Apple devices because it works with the Apple Find My network. Figure 6.1 shows what is it.

Figure 6.1: Chipolo one spot tag

6.2 Strategy to detect it in the air

The goal was to find a way to detect Chipolo trackers in the air, so as to filter the Wireshark capture to show only packets sent by it.

In the lab environment, there is a lot of Apple device, which send Apple Find My data. So, I had to find a way to clearly identify the Chipolo tag among other devices. To do that, a strategy could be to power off the iPhone when the tag is paired to its Apple ID account, and do a packet captures for 5 minutes[1]. During this capture, between 60 and 240 seconds, the tag's battery was removed.

[1] 300 seconds

Then, with the capture and the R language, it's easy to aggregate devices by MAC addresses. After that, to see what device has stopped to send BLE packets between 60 seconds and 240 seconds[2], it's a solution to calculate the time gap between the nth and the $(n-1)^{\text{th}}$ packet. Then, it's easy to plot, on the x axis, the time spent[3], and on the y axis, the time gap.

With this, there is an interesting plot in figure 6.2. There is a gap of 179 seconds for $t = 241$. It corresponds to the moment when the battery was pushed in the tag to power it on. It's coherent with the experimentation, because $240 - 60 = 180$.

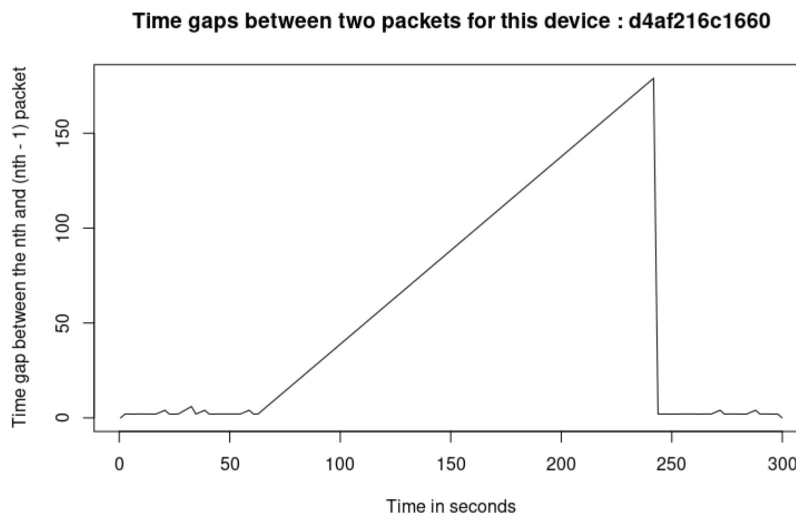

Figure 6.2: Time gap between two packets when the tag was unpowered between 60 sec. and 240 sec.

6.3 Experimentation to fake it with the Raspberry Pi

Now, to be sure it's a good way, it could be interesting to try to send the same data the Chipolo sends. To do that, it's important to remove its battery, and send its packets all at the same time intervals (Here, one packet by second for three minutes).

With a walk in the street with the Raspberry Pi powered by an external battery and sending Chipolo's packets and the original Chipolo disabled, we can test if effectively the MAC address was the good one or not.

Figure 6.3 shows the Apple Find My application on the iPhone after the walk. It tells the tag was seen for the last time at Bahnhof Oerlikon Ost[4] tramway station. So, the found device was the good one. The dark blue disk near "Universität Zürich - Psychologisches Institut" is the position of the iPhone which stayed at the lab during the experimentation.

[2]Duration of power off of Chipolo tag
[3]From 0 to 300
[4]Oerlikon train station east

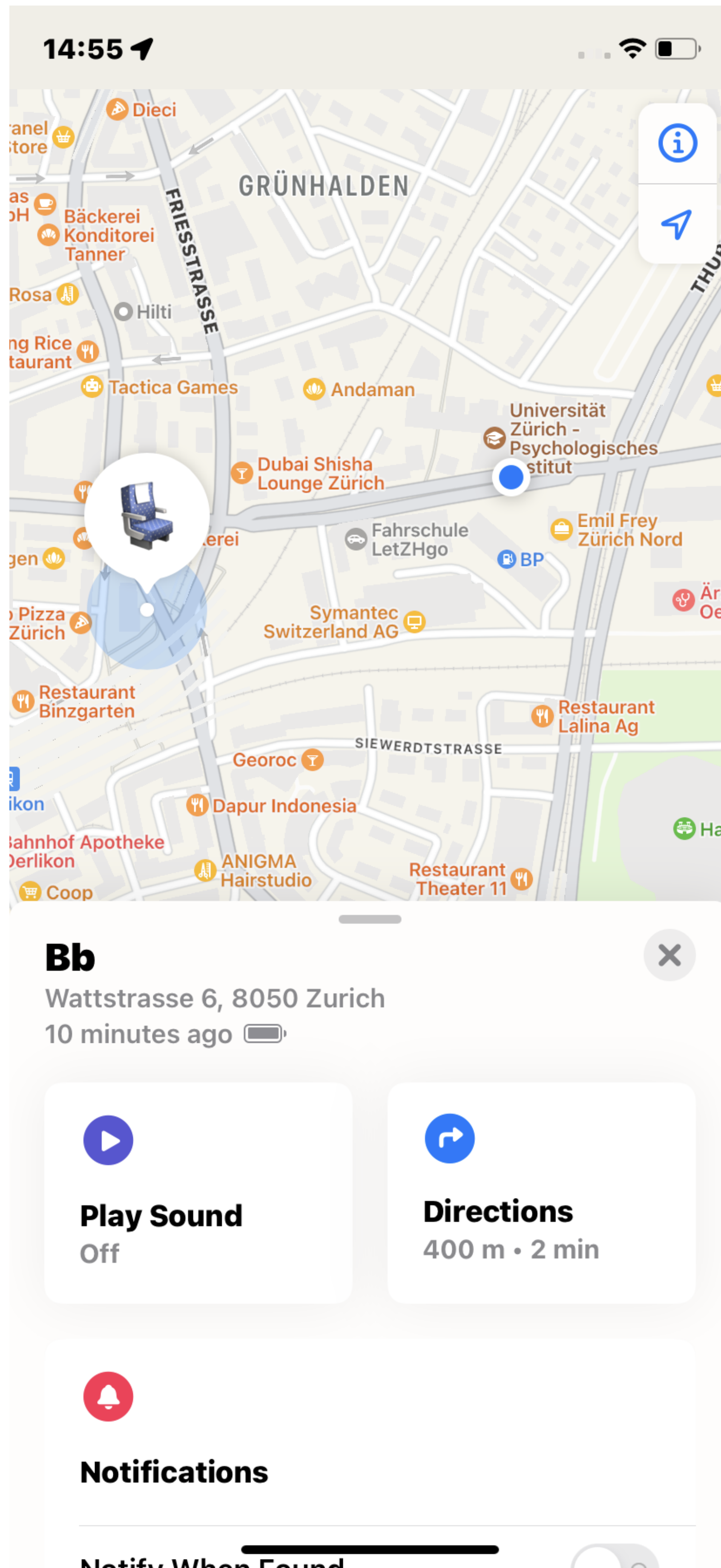

Figure 6.3: Result on Apple Find My application after walking in the street with the Raspberry Pi

6.4 Generalization

After that, it must be found a better way to see if an Apple BLE advertising packet is from a Chipolo tag or not. To do that, we need to collect more data about Chipolo tags, and a manner to do that is to disband it and pair it again to the Apple ID account associated with the iPhone.

Then, it was found an interesting thing: All of the Chipolo tags' manufacturer-specific advertising data always begins with `121920`. It corresponds to the beginning of the sequence found in the previous chapter[5].

After some tests, it turns out that is a good way to find Chipolo tags. To check that, it's necessary to make a new capture with the real tag powered on during all of that. Therefore, the Raspberry Pi has to be disabled. Then, with a filter that considers this parameter, Wireshark displays only packets from the Chipolo tag. It's trusted by doing the same experimentation as the previous section[6].

6.5 Summary

Finally, these characteristics let us distinguish if a sent packet is from a Chipolo One Spot tag, or not :

- The sent packet must have a "Manufacturer specific" field
- The length of this field must equal 30[7]
- The company ID of this field must be Apple's one[8]
- The data field begins by `121920`

[5] See chapter 5
[6] Section 6.3
[7] `1e` in hexadecimal
[8] `0x004c` in big-endian

Chapter 7

Samsung Galaxy SmartTag+

7.1 Introduction

The Samsung Galaxy SmartTag+ is another item-tracking device designed to work only with Samsung Galaxy phones. It's a Tile[1] competitor device.[12]

Like Tile and Apple AirTag, the Samsung Galaxy SmartTag+ uses an offline finding network, where every Samsung device, such as phones, tablets, etc. sends reports to Samsung's servers when it "sees" a device via BLE.

Figure 7.1 shows Samsung SmartTags. The SmartTag+ has the same visual as SmartTag. The difference between the two is the SmartTag+ uses UWB to estimate the distance between the tag and the phone and tries to find its location if it's possible.

[1]Described in chapter 4

Figure 7.1: Samsung Galaxy SmartTags

7.2 Analysis

To begin with, it's important to take note of the fact I haven't a Samsung Galaxy device when I made this analysis, so I was unable to pair it to a Samsung account.

But, with the technique used to detect the Chipolo OneSpot among other BLE devices[2], it's possible to detect the unpaired Samsung SmartTag+ among other devices.

Then, with that, there is a relevant device in the captures. All packets sent by this device contain these fields :

- A flags field
- An incomplete 16-bit Service Class UUIDs[3]
- The associated Service Data field for the UUID `0xfd59`

In the BLE 16-bit UUID reference, the UUID stands for "Samsung Electronics Co., Ltd.".[1] But, there are no more precise information about this, except that there are eight UUIDs for Samsung in the document.

[2] More details at section 6.2 of chapter 6

[3] Incomplete means the packet does not carry all of the supported UUIDs of the tag.

7.3 Summary

To conclude, it looks like packets sent by a Samsung SmartTag+ (and probably SmartTag) are crafted like this template, and it could be a way to see Samsung SmartTags around a generic phone.

Chapter 8

Summary and Conclusions

8.1 Summary

To conclude, during this internship, I worked on the privacy issue of item trackers. In fact, it's easy to hide a tag in a bag, in a car, etc. So, it was decided to develop a Flutter application to let users detect if there are BLE devices around them.

To find a way to detect those, it was used these kinds of tags :

- Apple AirTag
- Tiles
- OpenHaystack
- Chipolo One Spot
- Samsung Galaxy SmartTag+

8.2 How to detect each tags ?

8.2.1 Apple AirTag

The sent packet must match these points :

- The sent packet only contains one field: ”Manufacturer Specific” identified to Apple (`0x004c`)
- The value of $t = (d[2]\&\bar{3}0^{16}) >> 4$ equals 1 where d is the data embed by the ”Manufacturer Specific” field of advertising data of packet

8.2.2 Tiles

The sent packet must have these fields in this order :

- Flags
- 16 bits service class UUID
- 16 bits UUID service data

Then, the UUID of the Tile service is `0xfeed`.

8.2.3 OpenHaystack and Chipolo One Spot

A packet sent by a Chipolo One Spot tag has these characteristics :

- The sent packet has a "Manufacturer specific" field
- The length of this field equals 30 in decimal
- The company ID of this field is `0x004c` in big-endian
- The data field begins by `1219`[1]

Also, experimentation of OpenHaystack shows us there could be devices that are tracked by the Apple Find My network, even if those devices are not registered to an Apple ID account. So, handmade devices such as OpenHaystack "tags" are not shown on the Apple Find My application. This fact could be a privacy issue if someone tries to make an OpenHaystack tag with a chip.

8.2.4 Samsung Galaxy SmartTag+

All packets sent by the Samsung Galaxy SmartTag+ contain these fields :

- A flags field
- An incomplete 16-bit Service Class UUIDs
- The associated Service Data field for the UUID `0xfd59`

[1] In the case of Chipolo One Spot, it begins by `121920`.

8.3 Conclusion

With the information provided previously, it could be developed an application to detect and advise users there are foreign tags around them.

About this application, it could be interesting to let users add their tags, to avoid the risk of fake-negative tag detection.

To conclude, this internship was a very good experience for me, because I discovered the world of academic research, and doing a Ph.D. may interest me. Also, I discovered the BLE communications topic.

Bibliography

[1] 16-bit UUID Numbers Document.pdf. https://specificationrefs.bluetooth.com/assigned-values/16-bit%20UUID%20Numbers%20Document.pdf.

[2] AirTag. https://www.apple.com/airtag/.

[3] Apple AirTag Reverse Engineering - Adam Catley. https://adamcatley.com/AirTag.

[4] Bluetooth 5 variations complicate PHY testing - EDN. https://www.edn.com/bluetooth-5-variations-complicate-phy-testing/.

[5] Company Identifiers. https://www.bluetooth.com/specifications/assigned-numbers/company-identifiers/.

[6] Bluetooth Low Energy -It Starts with Advertising. https://www.bluetooth.com/blog/bluetooth-low-energy-it-starts-with-advertising/, February 2017.

[7] *Bluetooth specifications*. Core Specification Working Group, July 2021. Revision 5.3.

[8] AirGuard - AirTag tracking protection. https://github.com/seemoo-lab/AirGuard, August 2022. original-date: 2021-07-12T09:49:25Z.

[9] Cyclic redundancy check. https://en.wikipedia.org/w/index.php?title=Cyclic_redundancy_check&oldid=1095008993, June 2022. Page Version ID: 1095008993.

[10] OpenHaystack. https://github.com/seemoo-lab/openhaystack, August 2022. original-date: 2021-02-22T13:23:06Z.

[11] Mohammad Afaneh. How Bluetooth Low Energy Works: Advertisements (Part 1) | Novel Bits. https://novelbits.io/bluetooth-low-energy-advertisements-part-1/, April 2020.

[12] Dieter Bohn. Samsung's Galaxy SmartTag is a $29.99 Tile competitor. https://www.theverge.com/2021/1/14/22227621/samsung-galaxy-smarttag-price-release-date-tile-locator, January 2021.

[13] Monica Chin. Tile announces partnership with Intel to track missing PCs. https://www.theverge.com/2020/5/7/21250464/tile-intel-partnership-release-date-news-features-laptop-tracker-hp-dragonfly, May 2020.

[14] Trevor Long. Tile trackers get slim and sticky with new products for 2019. https://eftm.com/2019/10/tile-trackers-get-slim-and-sticky-with-new-products-for-2019-66642, October 2019.

Glossary

Apple AirTag Apple AirTag is a tracking device released by Apple in April 2021. It uses Bluetooth Low Energy (BLE) and Ultra Wide Band (UWB) to work. Because of it's small size, it can be used to do criminal actions, like stalking or thefting cars.. 1, 3, 9–11, 13, 15, 27

R R is a programming language to do some statistic analysis, and make some charts and visualizations. It could be used with RStudio, a software which help programmer to use the language.. 4, 11, 24

CRC Cyclic Redundancy Check (CRC) is an error-detecting code which is integrated in each packets to determine if the packet is good or not. If there are a CRC error, the packet can't be processed, and is rejected.. 4, 5

Apple ID An account stored in Apple's servers to centralize user's information betwenn Apple devices (Such as Mac with Mac OS/X, iPad, iPhone, etc.). 9, 11, 13, 21, 23, 26, 32

Apple Find My A network between Apple devices to ensure their tracking when one of these is lost, or theft.. 9–11, 13, 15, 17, 19, 21, 23–25, 32, G

NFC Near Field Communication (NFC) is a technology to communicate with a device by bringing it beghind another device (For example, a smartphone). The goal is the bringed device provides information, like a web address, to give informations about the device. Usually, it's a technology used in verry small devices, such as tracking tags. Moreover, it's the technology used by credit cards to pay without contact.. 10

MAC address Media Access Control (MAC) is a 48 bit address (6 bytes) used to identify a device. It should be unique in the world, but it can be not if the device uses a rendomly generated address or an user-provided address.. 10, 13, 24

SOC A System On a Chip (SOC) is a chip which contains everithing to run a firmware and work as a tiny computer : A CPU, inputs and outputs, RAM, etc. It can contains extra peripherals, like sensors, antenas, Read-Only Memory (ROM) to store a firmware, etc.. 17

hackintosh An hackintosh is a fake Apple Macintosh computer. In fact, it's a standard computer with Mac OS/X installed on it, the Macintosh's operating system. Finally, the word "hackintosh" is a combination of "Hack" and "Macintosh".. 17, 19, H, K

QEMU-KVM QEMU-Kernel-based Virtual Machine (KVM) is native virtualization technology which works on Linux-based operating systems. It can emulate lot of computer architectures.. 17, 18, G

Bash Bash, for Bourne-Again SHell, is a command-line interpreter for Unix systems (Like Linux). It's the default shell on most Linux systems, like Raspbian, the reference Linux system of Raspberry Pi. Then, it's also a programming language used to automatize some operation on theses systems.. 19

UWB Ultra Wide Band (UWB) is a wireless technology where devices send very short bursts of waves. Like BLE, it's a technology based on low energy consumtion. It can be used to estimate the distance between two devices by using signal propagation time. Therefore, with three UWB devices, it's possible to locate another UWB device by triangluation.. 27

firmware A firmware is an embedded software on a device to let it working. It can be updated to have new features, more security, etc.. C

Acronyms

CSG Comunication Systems Group. i

BLE Bluetooth Low Energy. iii, C, D, 1, 3–8, 10, 15, 17–19, 21, 24, 26–28, 31, 33, G

LL Link-Layer. iii, 5, 7

PDU Protocol Data Unit. iii, 5, 6, G

IOT Internet Of Things. 3

CRC Cyclic Redundancy Check. C, 4, 5, 12

UUID Universally Unique IDentifier. 6, 7, 16, 28, 32

UWB Ultra Wide Band. C, D, 9, 27

NFC Near Field Communication. C, 10

MAC Media Access Control. C, 10, 13, 24

SOC System On a Chip. C, 17

KVM Kernel-based Virtual Machine. D, 17, 18, G

CPU Central Processing Unit. C, 18

RAM Random Access Memory. C, 18

ROM Read-Only Memory. C

List of Figures

2.1 Ubertooth One USB key . . . 4

2.2 Diagram of BLE-LE packet . . . 5

2.3 Diagram of PDU header fields . . . 5

2.4 Diagram of fields of advertisement PDU payload . . . 6

3.1 An Apple AirTag . . . 9

3.2 Graph showing time gap between two packets when the iPhone is off . . . 12

3.3 Graph showing time gap acceleration between two packets when the iPhone is off . . . 12

3.4 Table showing number of packet sent during the capture "AirTagPairingAndPairedWhenIPhoneOn.pcapng" . . . 13

4.1 Example of Tile trackers . . . 15

5.1 Mac OS/X Big Sur (version 11) running on QEMU-KVM . . . 18

5.2 Raspberry Pi Zero W, the Raspberry Pi used to test OpenHaystack . . . 19

5.3 Test of OpenHaystack when walking in Oerlikon, at Zürich . . . 20

6.1 Chipolo one spot tag . . . 23

6.2 Time gap between two packets when the tag was unpowered between 60 sec. and 240 sec. . . . 24

6.3 Result on Apple Find My application after walking in the street with the Raspberry Pi . . . 25

7.1 Samsung Galaxy SmartTags . . . 28

A.1 Graph showing time gap between two packets when the iPhone is on . . . I

A.2 Graph showing time gap acceleration between two packets when the iPhone is on . . . . . . J

B.1 Raw reports on OpenHaystack in the hackintosh virtual machine . . . . . . K

Appendix A

Time diagrams of packet transmitted by when unpaired and iPhone on

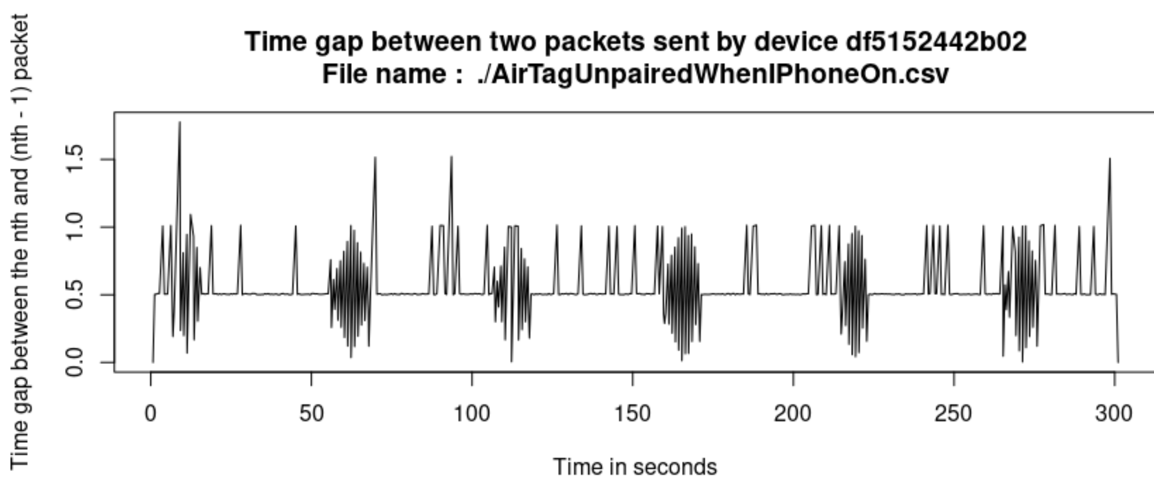

Figure A.1: Graph showing time gap between two packets when the iPhone is on

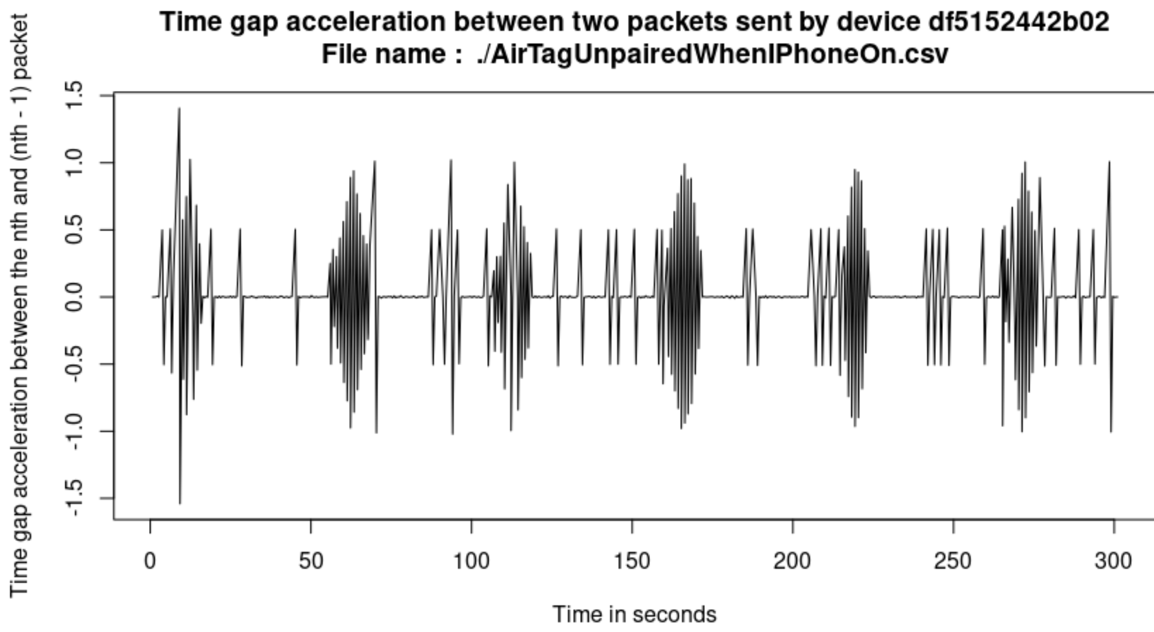

Figure A.2: Graph showing time gap acceleration between two packets when the iPhone is on

Appendix B

Generation of the picture of OpenHaystack reports

Because the used Mac system is not a genuine Macintosh, there are some limitations. And one of those is Apple map does not work. So, I had to superpose reports made by the walkers' iPhones on the google map by fitting pins with the taken route, and the result was accurate with the reality. The figure shows B.1 the real view on the application on the left, and the corresponding map on the right.

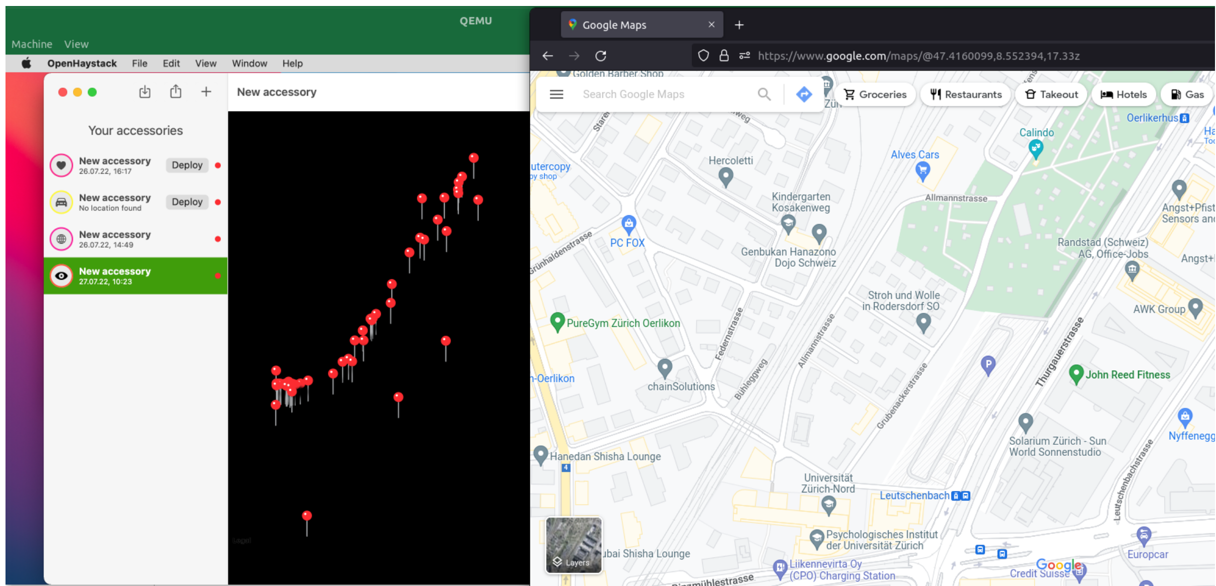

Figure B.1: Raw reports on OpenHaystack in the hackintosh virtual machine