TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Software Science

Stanislav Mekinulashvili 195997IVCM

SNIFFING ENCRYPTED BLE TRAFFIC AFTER CHANGING CONNECTION PARAMETERS, USING LOW-COST HARDWARE THAT CAPTURES ONLY ONE CHANNEL AT A TIME

Master's Thesis

Supervisor
Olaf Manuel Maennel
Ph.D. (Dr.rer.nat.)

Tallinn 2021

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutisüsteemide instituut

Stanislav Mekinulashvili 195997IVCM

BLE-LIIKLUSE ÜLESKIRJUTAMINE PÄRAST ÜHENDUSE PARAMEETRITE MUUTMIST, KASUTADES MADALA HINNAGA RIISTVARA, MIS VÕIMALDAB ÜLESKIRJUTADA KORRAGA AINULT ÜHTE KANALIT

Magistritöö

Juhendaja
Olaf Manuel Maennel
Ph.D. (Dr.rer.nat.)

Tallinn 2021

Acknowledgements

I would like to express my sincere gratitude to Dr. Olaf Manuel Maennel for helping me stay on my path. For his guidance, experience, encouragement, and support.
I would also like to extend my sincere thanks to Sultan Qasim Khan for helping me with identifying the research topic and for technical guidance throughout the work.

Author’s declaration of originality

Author:	Stanislav Mekinulashvili
Date:	14-05-2021

Abstract

Sniffing Bluetooth Low Energy communications has been historically problematic due to the Adaptive Frequency Hopping technique used by the protocol. Currently, the problem of AFH is solved either by expensive Software Defined Radios that capture the full RF spectrum or by low-cost hardware that has to acquire connection parameters and follow the AFH sequence.
This study focuses on acquiring connection parameters for encrypted connections after they change. The behavior of several most popular BLE stacks has been observed. Collected data has been visualized and analyzed. Using the collected data, a solution has been developed for connection parameter and PHY mode updates. The solution has been implemented on an open-source platform - Sniffle. This study also includes data that will be useful for future work on Channel Map updates.

The thesis is written in English and contains 67 pages of text, 5 chapters, 59 figures, 2 tables.

Annotatsioon

Bluetooth-i madala energiatarbega (ingl. Low Energy) side liikluse üleskirjutamine on olnud ajalooliselt problemaatiline protokolliga kasutatavale adaptiivse sagedusega hüppamise (ingl. Adaptive Frequency Hopping) tehnika pärast. Praegu lahendavad AFH probleemi kas kallima tarkvaraga määratletud raadiod (ingl. Software Defined Radios), mis hõivavad kogu raadiospektri, või odav riistvara, mis peab hankima ühenduse parameetrid ja järgima AFH järjestust.
Käesolev magistritöö keskendub krüpteeriitud ühenduste ühenduse parameetrite hankimisele pärast nende muutumist. Uuringu käigus on täheldatud mitme populaarseima BLE stack-i käitumist. Kogutud andmeid on visualiseeritud ja analüüsitud. Kasutades kogutud andmeid on välja töötatud lahendus ühenduse parameetrite ja PHY-režiimi värskendamiseks. Lahendus on rakendatud avatud lähtekoodiga platvormil - Sniffle. Käesolev magistritöö sisaldab ka andmeid, mis on kasulikud edasiste Channel Map tööde läbiviimiseks.

Lõputöö on kirjutatud Inglise keeles ning sisaldab teksti 67 leheküljel, 5 peatükki, 59 joonist, 2 tabelit.

List of abbreviations and terms

AA	Access Address
BT	Bluetooth
BLE	Bluetooth Low Energy
SIG	Special Interest Group
HCI	Host Controller Interface
GATT	Generic Attribute Profile
PHY	Physical Layer
IoT	Internet Of Things
PRNG	Pseudo-Random Number Generator
AFH	Adaptive Frequency Hopping
PDU	Protocol Data Unit
CSA	Channel Selection Algorithm
VM	Virtual Machine
RF	Radio Frequency
RSSI	Received Signal Strength Indicator

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Motivation

The number of smart devices with wireless connectivity is growing as the Internet of Things (IoT) gains prominence in many application domains [1]. With industry estimates predicting 5 billion Bluetooth-powered devices shipping in 2021 [2], Bluetooth technology will play a key role in the growing ecosystem of connected devices. The Bluetooth Special Interest Group (SIG) has implemented, beginning with version 4.0 of the Bluetooth Core Specification, a new version of the protocol called BLE, which is not backward-compatible with previous versions of the standard. For devices that have low data rate demands and strict power usage limits, BLE is best suited, thereby being the unwritten standard for many IoT applications. It is important to explore the security and privacy implications of BLE due to the growth of BLE-embedded IoT applications. The wireless sniffing attack is the leading attack on BLE computers, leading to more harmful attacks such as relay attack, jamming, encryption breaking, or system penetration.

Smart locks are a new advancement of the Internet of Things (IoT) that offers an access management solution that targets specific customers. They have been found to be susceptible to relay attacks against their Bluetooth communications [3, 4], among a number of other critical vulnerabilities commonly found in IoT devices. In particular, Bluetooth has been shown to be extremely vulnerable to relay attacks [5], partially due to its high and inconsistent latency, which decreases the efficiency of latency-based distance-bounding [6].

There is another common reason for capturing BLE traffic: troubleshooting and an alternative way to capture it: Host Controller Interface (HCI) snoop logging. One downside of HCI logging is that what is going over HCI is not the same as what is going over the air. Bluetooth controller performs several transformations, including encryption. Also, there is no visibility into lower-level processes. When you are trying to troubleshoot issues in a Bluetooth controller, it is useful to have the ability of low-level capture. A situation when over-the-air sniffing is the only solution is when two embedded devices, where you do not have much control over either of them, are communicating with each other over BLE.

1.2 Research problem

The problem with sniffing encrypted BLE traffic is that in order to successfully follow frequency hops and capture traffic using hardware that can sniff only one channel at a time, all connection parameters need to be known. These connection parameters do get updated periodically throughout an active connection. New parameters are encrypted, but the fact of exchanging new connection parameters before they get applied, can be identified. Unless new connection parameters are acquired, it is not possible to follow frequency hopping sequence, and accordingly, it is not possible to capture traffic either. An algorithm that can obtain new connection parameters is described in Mike Ryan's paper [7]. This algorithm is implemented in modern sniffers like BtleJack [8] and Ubertooth [9]. The algorithm is reliable but slow, since it is looking for new parameters almost from scratch, although usually, only a few parameters change. Ryan's algorithm is better suited for obtaining connection parameters for long-lived connections, rather than maintaining ability to follow frequency hopping on already established sniffing sessions.

1.3 Usage

1.3.1 Relay attack

This study is a step towards building a solution that is able to perform relay attacks on link layer. Existing solutions, like BtleJuice [10] and GATTacker [11], described in more detail later in the paper, are operating above the GATT layer, and it brings limitations to its usage. These attacks tend to have high latency, and they do not work with encrypted connections that have an unknown key. Performing a relay attack on link layer level will allow relaying packets without the need to understand their content, therefore allowing relaying encrypted packets, as well as unencrypted ones.

"In the relay attacks, adversary C talks to victim A posing as victim B, and to B posing as A. All authentication messages that C needs are generated by real A and B. C conveys these messages from A/B to B/A." [5]

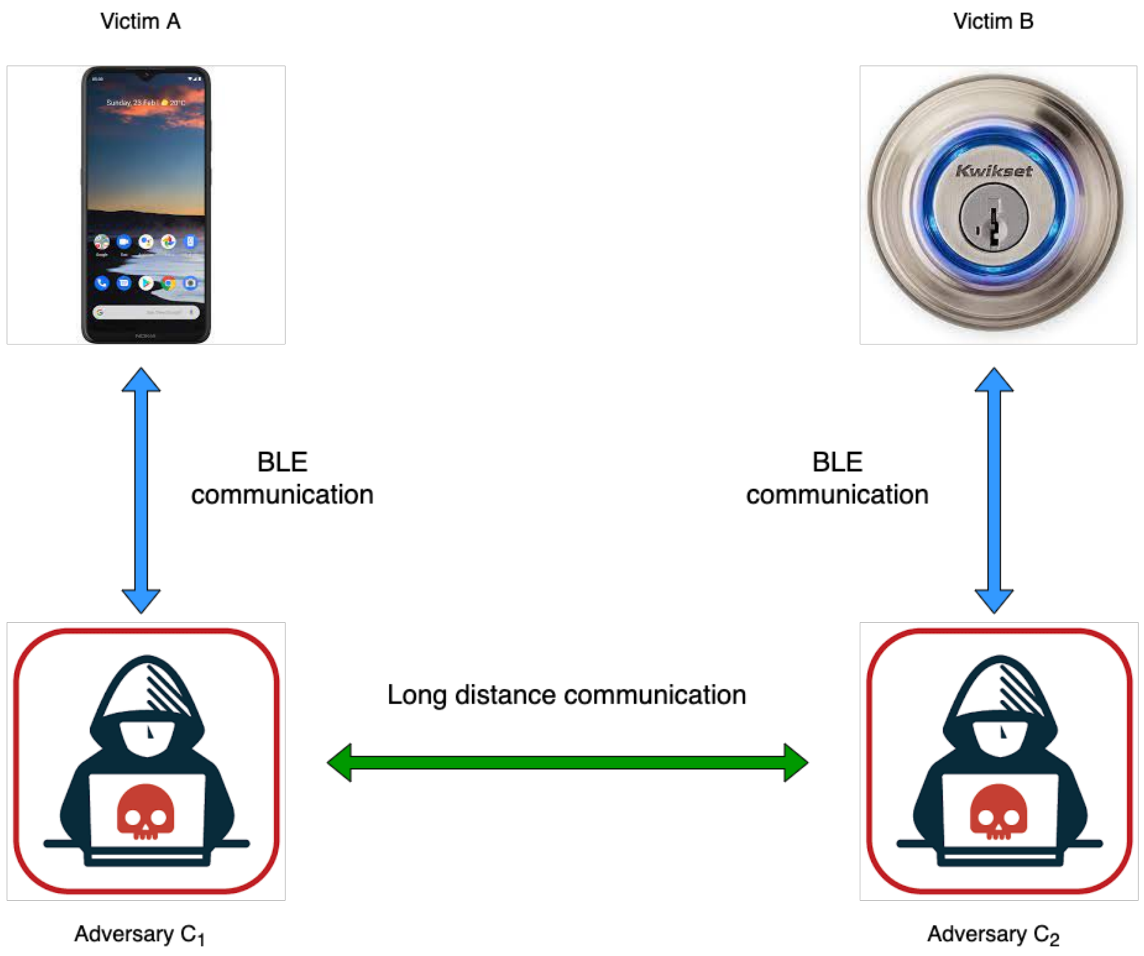


Figure 1. *Relay attack scheme on a smart-lock example*

1.3.2 Combining with Crackle

Crackle [12] is a tool by Mike Ryan that can, in some scenarios, extract Long Term Key (LTK) from captured pairing packets. Pairing act is usually followed by enabling encryption. LTK can later be used for decrypting captured data. Since encryption is enabled after pairing, it is crucial for a sniffer to be able to guess connection parameter changes. Otherwise, it will not be able to continuously capture encrypted data.

Another way of combining sniffing with Crackle would be trying to crack LTK without losing synchronization with the target and then decrypt the traffic on the fly using the obtained LTK. This approach can work, but in the time frame between enabling encryption and retrieving LTK, the communication content will still be unknown to the sniffer. As a result, when connection parameter update happens before Crackle can retrieve the LTK, synchronization will be lost. Ability to guess new parameters will increase reliability of such a solution.

1.4 Scope and contribution

This work explores the possibility of obtaining new connection parameters for encrypted connections after new connection parameters are exchanged on established sniffing sessions. This means that this paper will not be focused on obtaining the initial synchronization.

This study will cover the scenario when connection LTK is unknown, and we have a cheap - therefore physically limited hardware, that can sniff only one channel at a time.

The goal of this study is to research different popular Bluetooth stack implementations in hopes of finding any weaknesses or shortcuts that were taken by their developers.

The contribution of this study is twofold. First, it provides collected data and data analysis for different popular Bluetooth stack implementations. The data is both used in this study and will be useful for future studies addressing channel map update procedure. The collected data is related to connection parameter and channel map update procedures. Second, the study provides solutions for LL_CONNECTION_UPDATE_IND and LL_PHY_UPDATE_IND updates.

The study will cover connection parameter update and channel map update process for most widespread platforms: Android, iOS, Windows, Mac, and Linux.

1.5 Limitations

This study relies on control packet sizes to recognize them when their content is encrypted. It is thoroughly explained in chapter 2.2.
The limitation comes from BLE v5.2 protocol introducing LE Power Control features that brought LL_POWER_CONTROL_RSP and LL_POWER_CHANGE_IND packets that have the same 12-byte length as the LL_CONNECTION_UPDATE_IND packet that we are interested in.

BLE v5.2 was released in January 2020, but as of May 2021, it is not widespread yet. The latest iPhone 12 released in October 2020 has Bluetooth v5.0. The latest MacBooks have BT v5.0 as well. Also, author was not able to find a BT v5.2 USB dongle. BT v5.0 is prevailing among USB dongles on major online shopping platforms, with few supporting BT v5.1. Only a few android phones support BT v5.2 - hence it is not yet possible to determine how big of an obstacle the new power control features are. This issue will become relevant in a couple of years.

1.6 Research questions

- Is it possible to affect connection parameters by controlling the data that is being sent between two devices?
- Is it possible to identify data exchange patterns, that were able to trigger connection parameter updates, by observing the connection?
- Is it possible to affect channel mapping by producing RF noise?
- Is it possible to correlate Wi-Fi RF noise with BLE channel map?
- Which PHY modes are commonly used and what is the general pattern of switching between modes?
- How effective is the solution for recovering LL_CONNECTION_UPDATE_IND parameters?
- How effective is the solution for recovering LL_PHY_UPDATE_IND parameters?

1.7 Research methodology

The methodology used in this research involves mostly an observational approach. Various popular BLE stack implementation's natural behavior has been observed. The primary quantitative data has been collected using a custom extension written for Sniffle [13] platform described in the chapter 3.2. Data has later been visualized for further analysis using a custom script mentioned in chapter 3.3

Semi-controlled experiments have been set up in order to measure the influence of RF noise on BLE channel map. The results would have been more accurate if the experiments were conducted in a faraday cage environment, but the setup described in chapters 3.5.1 and 3.5.2 was enough to show strong dependency. Although, the imperfectness of environmental conditions has produced several artifacts, like in chapter 3.5.2

Data was collected by observing unencrypted communications that happen before two devices are bonded.

1.8 Ethics

Since over-the-air data sniffing is involved, author was especially careful to capture only the data that is considered public: e.g., advertisement data. Private communication data was captured using only devices that author owns, or has taken permission to use for capturing purposes.

2. Background

Forward and backward snowballing techniques were used during the literature review [14]. The initial set of literature was built by results of searches that included keywords: *BLE sniffing, Bluetooth sniffing, BLE Advanced Frequency Hopping, Bluetooth Advanced Frequency Hopping, BLE connection parameters, Bluetooth connection parameters, BLE connection parameter update, Bluetooth connection parameter update, BLE relay attack, Bluetooth relay attack*. For the initial set, Scopus [1] and Google Scholar [2] were used. It helped in avoiding potential bias when only looking at particular journals [14]. In addition to the set, papers describing officially recognized vulnerabilities listed on Bluetooth's web page were included [15].

The following chapters include grey literature sources in addition to academic papers. Because BLE sniffing is a theoretical as well as a highly practical field of study, grey literature is important for comprehensive literature analysis.

2.1 Literature review

When researching BLE - capturing traffic is one of the steps that have to be taken, and it has been historically problematic. In the Bluetooth world, capturing traffic is not as easy as launching Wireshark. Multiple solutions exist for capturing packets that are sent over the air [8, 9, 10]. "Several attempts were made to build a low-cost and open-source Bluetooth sniffer for over the air eavesdropping [9], [16], [17]. Unfortunately, an affordable and reliable solution is still not here." [18] "To reverse engineer the board's Bluetooth firmware, we load the dumped ROM, the symbols, the RAM, and the patch RAM regions into a Ghidra project" [18].

Antoniolli and his team, who have discovered more than half of the officially recognized vulnerabilities listed on Bluetooth's webpage [15], state the same in all of their research papers [18, 19, 20]. Mike Ryan calls Ubertooth [9] unreliable and says that the most reliable way to capture Bluetooth traffic is via HCI snooping. [21]

[1]Scopus `https://www.scopus.com/`
[2]Google Scholar `https://scholar.google.com/`

The sniffing problem comes from the physical limitations of hardware, which allows sniffing only one channel at a time.

Some older papers like “Bluetooth: With Low Energy comes Low Security” [7] and “A Robust Algorithm for Sniffing BLE Long-Lived Connections in Real-time” [22] are addressing connection parameter recovery problem, but they are focusing on doing it from scratch - when it is considered that nothing is known about the connection. Their methods are robust, but they can take a long time to complete - up to a minute. These methods are better suited for obtaining an initial synchronization with a connection rather than maintaining it. The goal of this study is to recover parameter changes within several hops. Depending on the hopping interval value, it would mean 30-300 milliseconds. Furthermore, these methods are using conclusions that are specific to CSA #1 specifications and will not apply to CSA #2, which is becoming a new standard since Bluetooth v5.

There have been several attempts to address the sniffing problem: “One GPU to Snoop Them All: a Full-Band Bluetooth Low Energy Sniffer” [23] is using SDR that can simultaneously track the traffic on all the 40 BLE channels and is out of the scope of our study.

AdaptaBLE [24] is a framework that describes ways of optimizing BLE connection parameters in a way that would ensure low power consumption while ensuring satisfactory quality of service. The author was not able to find out whether described principles are actually implemented by any vendor, but even if vendors use different optimization methods, the methods would be close to the ones described in the paper.

2.2 Relevant technical specifications

In the world of radio frequency-based devices, there is a relevant problem of wave interference. For Wi-Fi, it is solved by having 13 different channels and automatically choosing the least busy one. Since Wi-Fi routers are usually physically static and the picture of surrounding existing devices does not change frequently, it is a solution that proved itself to be reliable. However, in Bluetooth world, operation conditions are different. Although it operates in the same frequency band - the signal strength is significantly weaker than in Wi-Fi. Plus, devices are constantly moving around and constantly entering and leaving each other’s area of coverage. It creates a need for a different approach to solve the problem of wave interference. The solution for the problem that Bluetooth SIG came up with is the main problem of over-air Bluetooth sniffing, and it is Adaptive Frequency Hopping [25]. More advanced functions are proposed and integrated with the legacy AFH technology to improve the performance of AFH-based Bluetooth transmission [26, 27, 28, 29]

BLE operates in the 2.4 GHz ISM band at frequencies 2408 – 2483.5 MHz. The spectrum is divided into 40 channels, indexed from 0 to 39. The spacing between channels is 2 MHz. Channels 37, 38, and 39 are used for advertising and the other 37 channels are used for data transmission. Although advertising channel indexes go in a row, on a physical layer – channel 37 is located at the beginning of the designated range, channel 39 is in the end and channel 38 is around the middle of the range.

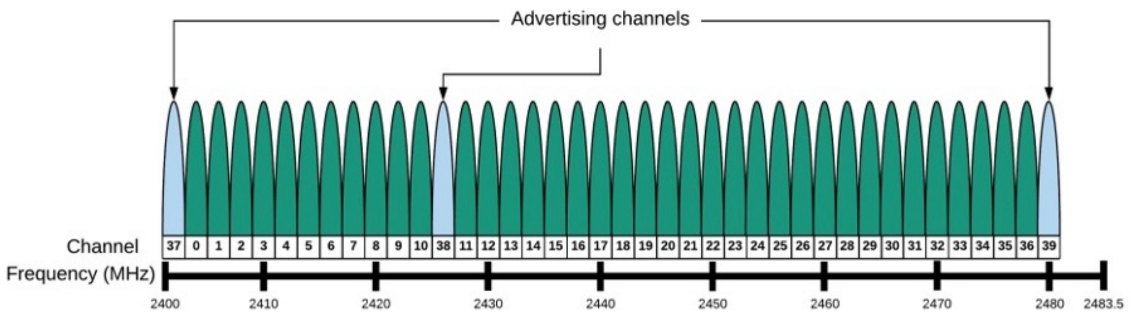


Figure 2. *RF channel scheme*
[30]

In essence, Frequency Hopping means that instead of choosing a less busy frequency and continuously operating on it, connection constantly switches from channel to channel, thereby decreasing the chance of interfering with other signals. In the first implementation of Bluetooth low energy, the channel selection algorithm was slightly more complicated than just increasing the current channel number by one. It was:

$$currentChannelNumber + x \pmod{37} \tag{2.1}$$

Where x is a variable connection parameter that is exchanged during pairing. Channel selection algorithm is responsible for generating a hopping sequence. Hopping sequence is a sequence of channel numbers that the connection will follow when hopping from channel to channel. The described algorithm is called Channel Selection Algorithm #1 (CSA #1) and is used in BLE v4.2 and below. Starting from BLE v5.0 - CSA #2 was introduced, which uses Pseudo-Random Number Generator and is more complicated than CSA #1

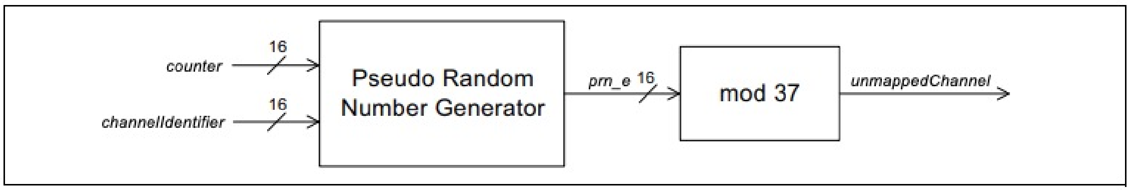


Figure 3. *Unmapped channel selection process*
[25]

As displayed in Figure 3, PRNG takes 2 arguments: counter and channel identifier as arguments and uses them as seeds, as is displayed in Figure 4, to generate a random number, that will later have modulo 37 applied to it. If not all channels are used - the result value goes through channel remapping process. CSA #2 generates a channel sequence that consists of 65535 channels which are being hopped circularly.

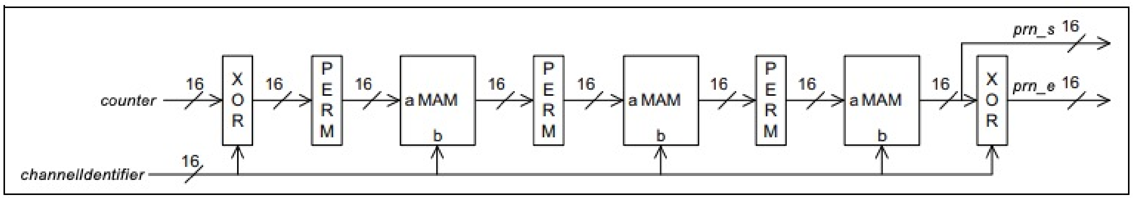
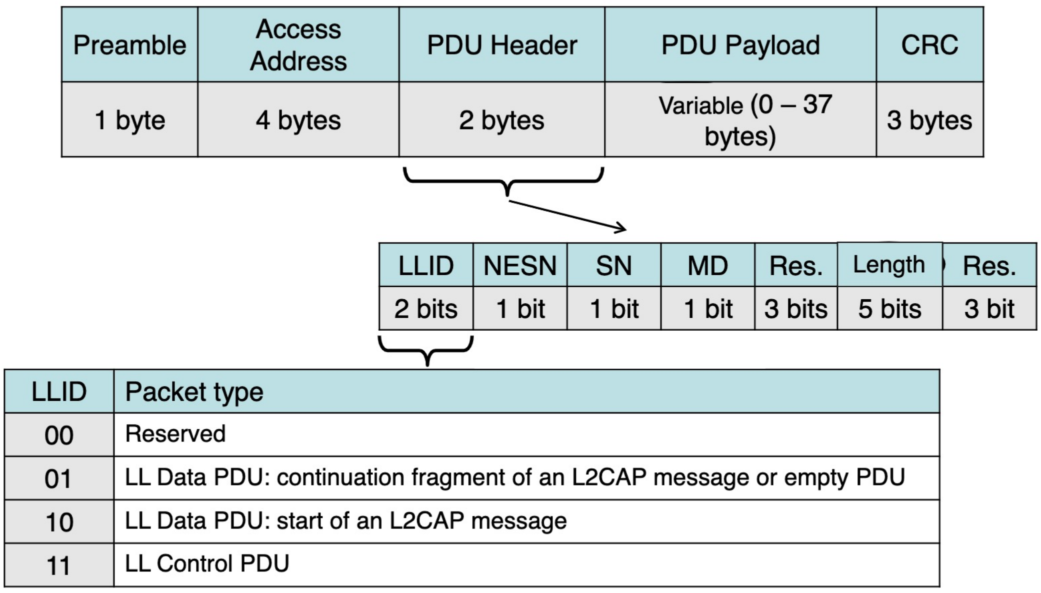


Figure 4. *Event pseudo-random number generation*
[25]

The passed argument named Counter is sometimes referred to in BT specs as Connection Event Counter, and sometimes just counter, but it represents the same concept. It represents the current position in the hopping sequence. In the current context, Connection Event can be viewed as a synonym to hop, since it represents an act of exchanging empty PDUs between master and slave devices after they both hop to the next channel in a sequence.

Channel Identifier is first 16 bits of AA (Access Address) XOR-ed with last 16 bits: "The 16-bit input channel Identifier is fixed for any given connection or periodic advertising train; it is calculated from the Access Address by: channelIdentifier = (Access Address $_{31\text{-}16}$) XOR (Access Address $_{15\text{-}0}$)" [25]

AA is a part of a Link Layer data PDU (Figure 5) and it is a randomly generated 32-bit value that substitutes source and destination mac addresses, that would be 96 bits in total. AA represents a connection. Devices involved in a connection determine if a packet is addressed to them according to AA, instead of a mac address.



Preamble	Access Address	PDU Header	PDU Payload	CRC
1 byte	4 bytes	2 bytes	Variable (0 – 37 bytes)	3 bytes

LLID	NESN	SN	MD	Res.	Length	Res.
2 bits	1 bit	1 bit	1 bit	3 bits	5 bits	3 bit

LLID	Packet type
00	Reserved
01	LL Data PDU: continuation fragment of an L2CAP message or empty PDU
10	LL Data PDU: start of an L2CAP message
11	LL Control PDU

Figure 5. *Link Layer: data PDU*

As Damien Cauquil has mentioned in his DEF CON talk [31], unlike other PRNG number generators, that use some hidden internal device state-related number as a seed, the seed of CSA #2 PRNG - Channel Identifier - can be considered public, since it is derived from Access Address, which is transmitted publicly with every packet and which never gets encrypted.

The minimal number of connection parameters that are required to be known to follow a connection are:

- Connection Interval
- Connection Event Counter
- Channel Map
- PHY mode

"Connection interval is the amount of time between two connection events in units of 1.25 ms. The connection interval can range from a minimum value of 6 (7.5 ms) to a maximum of 3200 (4.0 s)" [25]

PHY mode - BLE 5.0 standard introduced 2 new physical modes on top of previously existed "LE 1M PHY": **LE 2M PHY** and **LE Coded PHY**

Channel Map - Though there are 37 data channels available, not all of them are used all the time. Devices may decide to stop using one or several ranges of channels and transfer data only through the ones that are left. Channel map describes which channels will be used during the connection.

There are also other important connection parameters that are worth mentioning but do not constitute a significant obstacle for sniffing: Connection timeout and Slave Latency.

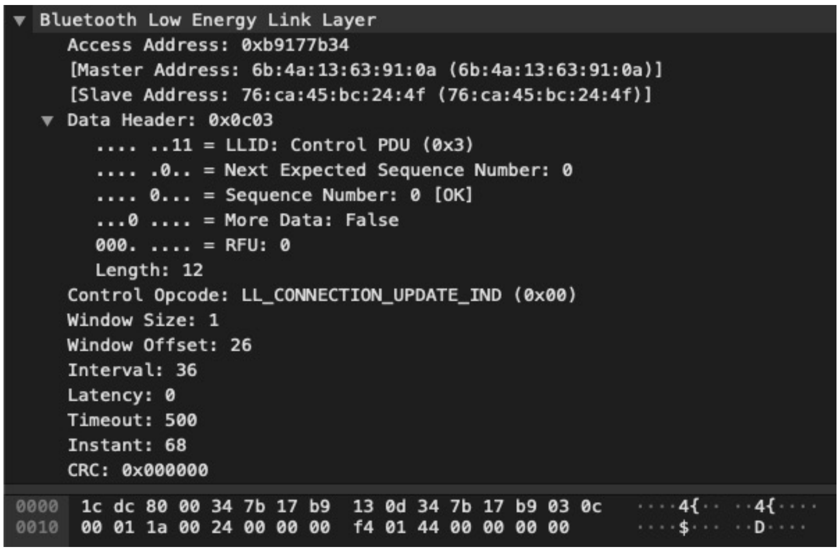
Slave Latency allows a slave to use a reduced number of connection events. It defines the number of consecutive connection events that the slave device is not required to listen for the master. It allows a slave device to skip several empty PDU exchanges for power-saving purposes, without losing a connection with the master device.

Connection Timeout describes time without a valid packet received from a slave device, in order for the connection to be considered lost.

Connection parameters listed above do get updated throughout a connection. The connection parameter update procedure involves several other parameters that do not affect an established connection, but are required to know in order to follow the connection update.

- Instant
- Window offset
- Window size

Connection parameters get updated through LL Control PDUs. There are 37 of them in BLE 5.0 and they are used for controlling different technical aspects of connection, but we are interested in 3 of them: LL_CONNECTION_UPDATE_IND, LL_CHANNEL_MAP_IND, and LL_PHY_UPDATE_IND.



```
▼ Bluetooth Low Energy Link Layer
    Access Address: 0xb9177b34
    [Master Address: 6b:4a:13:63:91:0a (6b:4a:13:63:91:0a)]
    [Slave Address: 76:ca:45:bc:24:4f (76:ca:45:bc:24:4f)]
  ▼ Data Header: 0x0c03
      .... ..11 = LLID: Control PDU (0x3)
      .... .0.. = Next Expected Sequence Number: 0
      .... 0... = Sequence Number: 0 [OK]
      ...0 .... = More Data: False
      000. .... = RFU: 0
      Length: 12
    Control Opcode: LL_CONNECTION_UPDATE_IND (0x00)
    Window Size: 1
    Window Offset: 26
    Interval: 36
    Latency: 0
    Timeout: 500
    Instant: 68
    CRC: 0x000000

0000  1c dc 80 00 34 7b 17 b9  13 0d 34 7b 17 b9 03 0c   ····4{·· ··4{····
0010  00 01 1a 00 24 00 00 00  f4 01 44 00 00 00 00      ····$··· ··D····
```

Figure 6. *LL_CONNECTION_UPDATE_IND packet example*

As we can see in Figure 6, LL_CONNECTION_UPDATE_IND has all 3 parameter-update-specific parameters.

Instant represents a Connection Event Counter, when devices should apply the new parameters **Window size** represents a period of time during which the first packet of connection with new parameters should be sent, and **Window offset** represents the length of a pause, that has to be taken between reaching the Instant and Transmit Window (Figure 7)

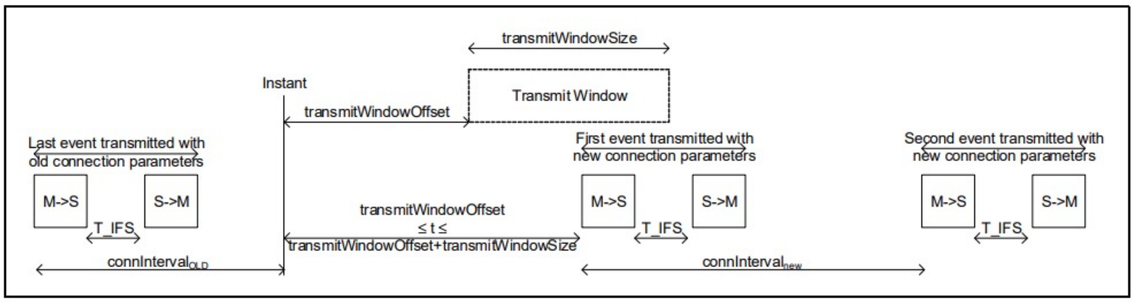
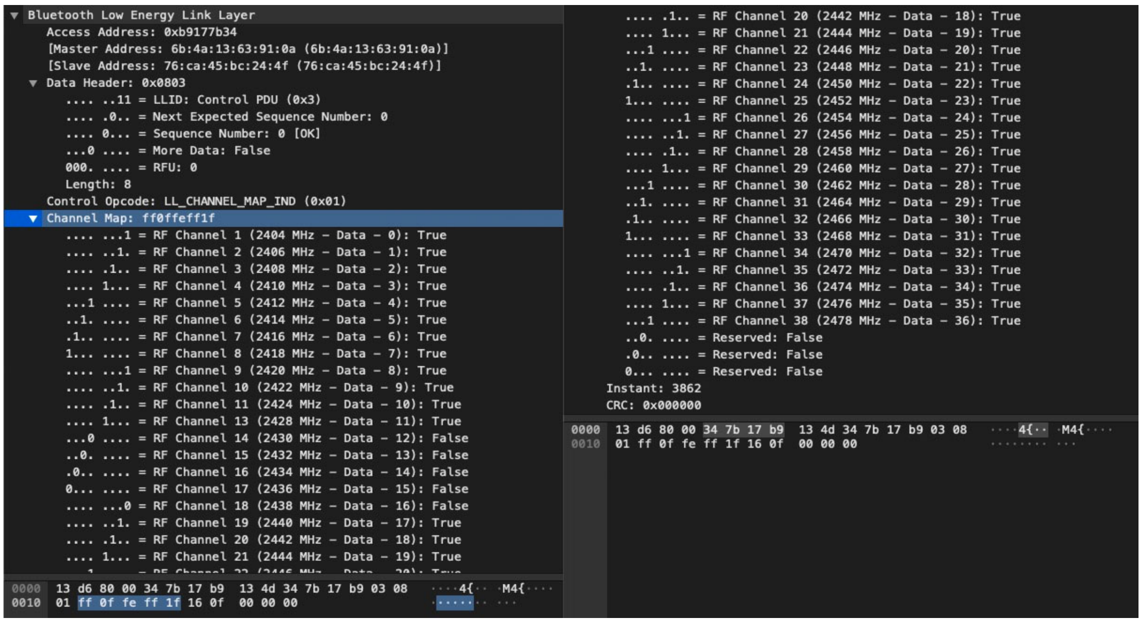

transmitWindowSize
Instant
Transmit Window
transmitWindowOffset
Last event transmitted with old connection parameters
First event transmitted with new connection parameters
Second event transmitted with new connection parameters
M->S
S->M
T_IFS
M->S
S->M
T_IFS
M->S
S->M
T_IFS
transmitWindowOffset ≤t≤ transmitWindowOffset+transmitWindowSize
connInterval$_{OLD}$
connInterval$_{new}$

Figure 7. *Connection event timing in the case of connection parameter update* [25]

LL_CHANNEL_MAP_IND (Figure 8) and LL_PHY_UPDATE_IND (Figure 9) only have Instant.

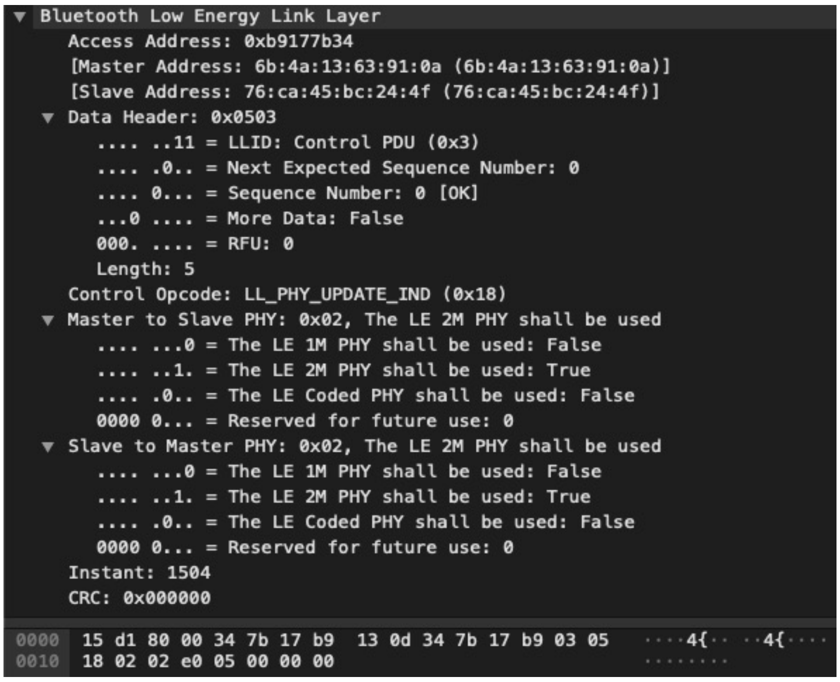

```
▼ Bluetooth Low Energy Link Layer
    Access Address: 0xb9177b34
    [Master Address: 6b:4a:13:63:91:0a (6b:4a:13:63:91:0a)]
    [Slave Address: 76:ca:45:bc:24:4f (76:ca:45:bc:24:4f)]
  ▼ Data Header: 0x0803
      .... ..11 = LLID: Control PDU (0x3)
      .... .0.. = Next Expected Sequence Number: 0
      .... 0... = Sequence Number: 0 [OK]
      ...0 .... = More Data: False
      000. .... = RFU: 0
      Length: 8
    Control Opcode: LL_CHANNEL_MAP_IND (0x01)
  ▼ Channel Map: ff0ffeff1f
      .... ...1 = RF Channel 1 (2404 MHz - Data - 0): True
      .... ..1. = RF Channel 2 (2406 MHz - Data - 1): True
      .... .1.. = RF Channel 3 (2408 MHz - Data - 2): True
      .... 1... = RF Channel 4 (2410 MHz - Data - 3): True
      ...1 .... = RF Channel 5 (2412 MHz - Data - 4): True
      ..1. .... = RF Channel 6 (2414 MHz - Data - 5): True
      .1.. .... = RF Channel 7 (2416 MHz - Data - 6): True
      1... .... = RF Channel 8 (2418 MHz - Data - 7): True
      .... ...1 = RF Channel 9 (2420 MHz - Data - 8): True
      .... ..1. = RF Channel 10 (2422 MHz - Data - 9): True
      .... .1.. = RF Channel 11 (2424 MHz - Data - 10): True
      .... 1... = RF Channel 13 (2428 MHz - Data - 11): True
      ...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
      0... .... = RF Channel 17 (2436 MHz - Data - 15): False
      .... ...0 = RF Channel 18 (2438 MHz - Data - 16): False
      .... ..1. = RF Channel 19 (2440 MHz - Data - 17): True
      .... .1.. = RF Channel 20 (2442 MHz - Data - 18): True
      .... 1... = RF Channel 21 (2444 MHz - Data - 19): True

0000  13 d6 80 00 34 7b 17 b9  13 4d 34 7b 17 b9 03 08   ····4{·· ·M4{····
0010  01 ff 0f fe ff 1f 16 0f  00 00 00                  ········ ···
```

```
      .... .1.. = RF Channel 20 (2442 MHz - Data - 18): True
      .... 1... = RF Channel 21 (2444 MHz - Data - 19): True
      ...1 .... = RF Channel 22 (2446 MHz - Data - 20): True
      ..1. .... = RF Channel 23 (2448 MHz - Data - 21): True
      .1.. .... = RF Channel 24 (2450 MHz - Data - 22): True
      1... .... = RF Channel 25 (2452 MHz - Data - 23): True
      .... ...1 = RF Channel 26 (2454 MHz - Data - 24): True
      .... ..1. = RF Channel 27 (2456 MHz - Data - 25): True
      .... .1.. = RF Channel 28 (2458 MHz - Data - 26): True
      .... 1... = RF Channel 29 (2460 MHz - Data - 27): True
      ...1 .... = RF Channel 30 (2462 MHz - Data - 28): True
      ..1. .... = RF Channel 31 (2464 MHz - Data - 29): True
      .1.. .... = RF Channel 32 (2466 MHz - Data - 30): True
      1... .... = RF Channel 33 (2468 MHz - Data - 31): True
      .... ...1 = RF Channel 34 (2470 MHz - Data - 32): True
      .... ..1. = RF Channel 35 (2472 MHz - Data - 33): True
      .... .1.. = RF Channel 36 (2474 MHz - Data - 34): True
      .... 1... = RF Channel 37 (2476 MHz - Data - 35): True
      ...1 .... = RF Channel 38 (2478 MHz - Data - 36): True
      ..0. .... = Reserved: False
      .0.. .... = Reserved: False
      0... .... = Reserved: False
    Instant: 3862
    CRC: 0x000000

0000  13 d6 80 00 34 7b 17 b9  13 4d 34 7b 17 b9 03 08   ····4{·· ·M4{····
0010  01 ff 0f fe ff 1f 16 0f  00 00 00                  ········ ···
```

Figure 8. *LL_CHANNEL_MAP_IND packet example*


```
▼ Bluetooth Low Energy Link Layer
    Access Address: 0xb9177b34
    [Master Address: 6b:4a:13:63:91:0a (6b:4a:13:63:91:0a)]
    [Slave Address: 76:ca:45:bc:24:4f (76:ca:45:bc:24:4f)]
  ▼ Data Header: 0x0503
      .... ..11 = LLID: Control PDU (0x3)
      .... .0.. = Next Expected Sequence Number: 0
      .... 0... = Sequence Number: 0 [OK]
      ...0 .... = More Data: False
      000. .... = RFU: 0
      Length: 5
    Control Opcode: LL_PHY_UPDATE_IND (0x18)
  ▼ Master to Slave PHY: 0x02, The LE 2M PHY shall be used
      .... ...0 = The LE 1M PHY shall be used: False
      .... ..1. = The LE 2M PHY shall be used: True
      .... .0.. = The LE Coded PHY shall be used: False
      0000 0... = Reserved for future use: 0
  ▼ Slave to Master PHY: 0x02, The LE 2M PHY shall be used
      .... ...0 = The LE 1M PHY shall be used: False
      .... ..1. = The LE 2M PHY shall be used: True
      .... .0.. = The LE Coded PHY shall be used: False
      0000 0... = Reserved for future use: 0
    Instant: 1504
    CRC: 0x000000

0000  15 d1 80 00 34 7b 17 b9  13 0d 34 7b 17 b9 03 05   ····4{·· ··4{····
0010  18 02 02 e0 05 00 00 00                            ········
```

Figure 9. *LL_PHY_UPDATE_IND packet example*

When connection gets encrypted, only the PDU payload (Figure 5) gets encrypted, and Access Address and PDU header remain unchanged. PDU header contains LLID that indicates packet type and lets us tell if it is a data PDU or control PDU.

In the Bluetooth 5.2 core spec [25], there are 37 different control PDU types defined. They vary in the number of parameters and their sizes. Their payload size, that is publicly transmitted in the PDU header, can become a mean of packet type identification: payload length of 5 bytes belongs only to LL_PHY_UPDATE_IND, 8 byte length is unique for LL_CHANNEL_MAP_IND, but 12 byte length as well as LL_CONNECTION_UPDATE_IND can also be representing LL_POWER_CONTROL_RSP and LL_POWER_CHANGE_IND. Since power control features were added in BT v5.2 and as mentioned in chapter 1.5, BT v5.2 is not widespread - this study will not take existence of related packets into account.

When the connection is encrypted, 4 byte MIC value is added to the payload, so every control PDU's length will be increased by 4 bytes in the encrypted mode, compared to unencrypted mode.

2.3 Existing solutions

2.3.1 Ubertooth

Ubertooth [9] is the most popular Bluetooth sniffing solution at the moment of writing this paper. It has ~1450 stars on Github, as of March 2021, and the last commit to the master branch was done on December 2020.
Ubertooth One is an open-source 2.4 GHz wireless development platform designed for Bluetooth experimentation and hacking. It was created by Michael Ossmann and Dominic Spill from Great Scott Gadgets [3]. It is considered to be one of the most capable Bluetooth network sniffing, real-time traffic monitoring and penetration testing platform, because it is the only low-cost sniffer that supports Bluetooth Classic.

Architecture

- RP-SMA RF connector: connects to test equipment, antenna, or dummy load
- CC2591 RF front end
- CC2400 wireless transceiver
- LPC175x ARM Cortex-M3 microcontroller with Full-Speed USB 2.0
- USB A plug: connects to host computer running Kismet or other host code [9]

Features

- 2.4 GHz transmit and receive
- Transmit power and receive sensitivity comparable to a Class 1 Bluetooth device
- Standard Cortex Debug Connector (10-pin 50-mil JTAG)
- In-System Programming (ISP) serial connector
- Expansion connector: intended for inter-Ubertooth communication or other future uses six indicator LEDs [9]

Ubertooth's popularity results not only in the biggest numbers of stars on Github, but it also is frequently used in BLE related projects and researches, like "A Robust Algorithm for Sniffing BLE Long-Lived Connections in Real-Time" [22], where they use 2 devices at the same time, or "Revisiting Bluetooth Adaptive Frequency Hopping Prediction with a Ubertooth" [32]
In "Practical Bluetooth Traffic Sniffing: Systems and Privacy Implications" [17] researchers are also simultaneously using 2 Ubertooths.

[3]Great Scott Gadgets `https://greatscottgadgets.com/`

Another indicator that Ubertooth may be developer-friendly is existence of major firmware extensions, like Bluetooth Low Energy Multi (BLE-Multi) [33] "It is a firmware extension to Ubertooth One which enables tracking of multiple simultaneous long-lived connections. It uses transmission of empty packets to determine the anchor point of each connection event and connection timing. Moreover, it achieves multi-connection sniffing by opportunistically switching between connections when they move from active to sleep mode." [22]

License type: GNU General Public License v2.0

Price with shipping to Estonia: From €130 on `amazon.de`, to €230 in local shops

2.3.2 BtleJack

BtleJack [8] is also a very popular sniffing platform that has ~1460 stars on Github and the last commit to the master branch was done in October 2019, as of March 2021.
"Btlejack provides everything you need to sniff, jam and hijack Bluetooth Low Energy devices. It relies on one or more BBC Micro:Bit.[4] devices running a dedicated firmware. You may also want to use an Adafruit's Bluefruit LE sniffer [5] or a nRF51822 Eval Kit, as we added support for these devices."[8]
"Current version of this tool (2.0) supports BLE 4.x and 5.x. The BLE 5.x support is limited, as it does only support the 1Mbps Uncoded PHY and does not support channel map updates." [8]

BtleJack can not only sniff, but also take control of a link by actively disconnecting the master and taking its place in the connection.

Features

- Sniffing an existing BLE connection
- Sniffing new BLE connections
- Jamming an existing BLE connection
- Hijacking an existing BLE connection
- Exporting captured packets to various PCAP formats [8]

License type: MIT License

[4]Micro:Bit `https://microbit.org/`
[5]Adafruit's Bluefruit LE sniffer `https://www.adafruit.com/product/2269`

Price with shipping to Estonia: €23 for Micro:Bit v2 on `lemona.ee` and €50 on `amazon.de` for Adafruit's Bluefruit LE sniffer

2.3.3 BtleJuice

BtleJuice framework [10] was developed by Damien Cauquil, who is the author of BtleJack, but it has some major differences with BtleJack.
BtleJuice does not use any specialized hardware and requires 2 Bluetooth 4.0+ adapters to operate.

Cauquil, in his talk [34] at GreHack, explains how it works and describes the concept of the framework to be close to how HTTP(S) proxies, such as Burp Suite or OWASP Zap, operate.
BtleJuice, in essence, simulates master device for the slave device and vice-versa, so that target devices would think that they are talking to each other. It establishes 2 separate connections and forwards the exchanged data to one another, capturing it in between.
According to Cauquil [34], one of the biggest drawbacks of the method is that Linux Bluetooth stack does not fully support more than 1 Bluetooth adapter functioning at the same time, so he proposes using another machine, or VM to overcome this issue.

BtleJuice has ~530 stars on Github, and the last commit to the master branch was done in October 2018, as of March 2021

License: without limitation, the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies.

2.3.4 GATTacker

GATTacker [11] is a tool by Slawomir Jasek and is very similar to BtleJuice. It similarly uses 2 Bluetooth 4 adapters, impersonating master and slave device for target devices and essentially transferring data between Bluetooth adapters via WebSockets.

GATTacker offers a possibility to intercept and modify traffic between adapters via web intercepting proxies like Burp Suite. The requests are being transferred in JSON format.

"Currently the tool works for devices which do not implement Bluetooth LE link-layer pairing/encryption. However there is surprisingly lot of such devices.
Possible attacks against encrypted connections are described in whitepaper [35]" [11]

GATTacker has ~500 stars on Github and the last commit to the master branch was done in October 2018, as of March 2021

License type: MIT License

2.3.5 Sniffle

Sniffle [13] is a tool by Sultan Qasim Khan that is based on Texas Instruments' TI CC1352/CC26x2 hardware that supports the newest features of Bluetooth 5.0+

Some of Sniffle's key features are:

- Support for BT5/4.2 extended length advertisement and data packets
- Support for BT5 Channel Selection Algorithms #1 and #2
- Support for all BT5 PHY modes (regular 1M, 2M, and coded modes)
- Support for channel map, connection parameter, and PHY change operations
- Support for BT5 extended advertising (non-periodic)
- Support for capturing advertisements from a target MAC on all three primary advertising channels using a single sniffer. This makes connection detection nearly 3x more reliable than most other sniffers that only sniff one advertising channel. [13]

Sniffle has ~400 stars on Github, and the last commit to the master branch was done in October 2020, as of March 2021

License type: GNU General Public License v3

Price with shipping to Estonia: €60 for CC26x2

3. Data collection

3.1 Choosing a platform

When looking for a platform to perform the research on, it soon becomes apparent that most of the existing solutions are becoming outdated due to new BLE 5 specifications.

BLE 5 brought several new features to the physical layer and many microcontrollers that are used by various low-cost sniffers, simply do not support them. Unless they are ported to newer hardware, they will not be able to implement BLE 5 support to the fullest. Given the fact that BLE 5 has been out for more than 4 years, and that porting software to a different microcontroller architecture can be tricky, and the fact that it has not been ported yet – the best option would be to go for a platform that currently supports all new features. Sniffle [13] not only has the biggest hardware support, compared to other options, but also has the biggest number of BLE 5 compliant features implemented software-wise.
Sniffle supports channel map updates for CSA #2, while BtleJack does not [8].
Proxying solutions (BtleJuice and GATTacker) work poorly with devices that have encryption enabled. They also will not allow tapping into long-lived connections.

Table 1. *Comparison to Low-Cost Sniffers* [36]

	Sniffle	Ubertooth	TI Sniffer v1 (CC2540)	TI Sniffer v2 (CC26xx)	Nordic nRF51	Nordic nRF52	BtleJack 2
Cost	$40	$120	$40	$40	$40	$40	$15
Open source	Yes	Yes	No	Yes	No	No	Yes
Data Length Extension	Yes	Yes	No	Yes	Yes	Yes	Yes
BT5 PHY	Yes	No	No	No	No	2M	No
BT5 CSA #2	Yes	No	No	No	No	No	Yes
BT5 Ext. Adv	Yes	No	No	No	No	No	No
Sniff 37/38/39	Yes	No	No	No	No	No	No
Tapping into existing connections	No (but can be implemented)	Yes	No	No	No	No	Yes

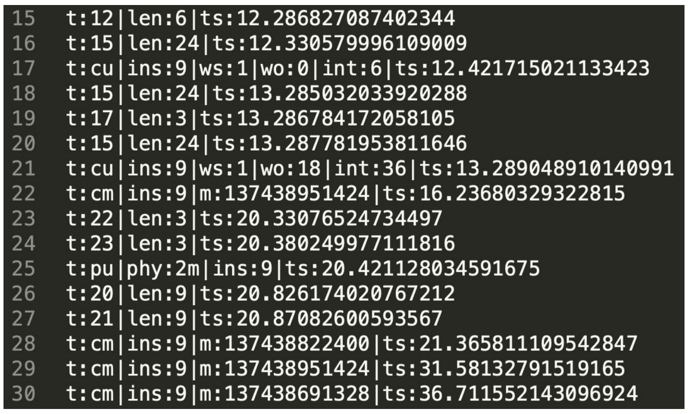
Table 2. *Comparison to Commercial Sniffers* [36]

	Sniffle	Ellisys Bluetooth Tracker Pro	Frontline BPA LE	Frontline Sodera LE WB
Cost	$40	From $10 000	$5 000	$10 000
Open source	Yes	No	No	No
Data Length Extension	Yes	Yes	Yes	Yes
BT5 PHY	Yes	Yes	No	Yes
BT5 CSA #2	Yes	Yes	No	Yes
BT5 Ext. Adv	Yes	Yes	No	Yes
Sniff 37/38/39	Yes	Yes (full SDR)	Yes (simultaneous)	Yes (full SDR)

3.2 Setup

Texas Instruments' CC2652R1 microcontroller was obtained to be used for this research, and Sniffle was installed on it according to the instruction [13] on GitHub. During the software setup, no issue has arisen that the said instruction did not cover.

When looking for a method for data processing, at first, pcap file parsing was considered since Sniffle already supports dumping captured data to pcap, but it soon became apparent that it would not work.
One of the important parameters - Instant - is exchanged and therefore dumped as an index number on the hopping sequence. The number we need to know is the difference between the current connection event counter and the instant (Ex: on figure 9 we can see that instant is 1504, and it points the devices at a particular hop number, where the update should occur. The counter value would be around 1495 at that particular moment). The counter value is a part of internal state of the devices that communicate, and is never transferred over the air. Sniffle stores the counter value as well, since it is a vital parameter and can calculate the difference. It was decided to add a small extension to Sniffle's code that would log relevant parameters from all captured control PDUs to a separate text file. Example of the captured data can be seen on Figure 10. The code can be found on the following link: `https://github.com/stas-me/Sniffle/commit/765e149961cc3505f49bca32d1b96e5880886930`



```
t:12|len:6|ts:12.286827087402344
t:15|len:24|ts:12.330579996109009
t:cu|ins:9|ws:1|wo:0|int:6|ts:12.421715021133423
t:15|len:24|ts:13.285032033920288
t:17|len:3|ts:13.286784172058105
t:15|len:24|ts:13.287781953811646
t:cu|ins:9|ws:1|wo:18|int:36|ts:13.289048910140991
t:cm|ins:9|m:137438951424|ts:16.23680329322815
t:22|len:3|ts:20.33076524734497
t:23|len:3|ts:20.380249977111816
t:pu|phy:2m|ins:9|ts:20.421128034591675
t:20|len:9|ts:20.826174020767212
t:21|len:9|ts:20.87082600593567
t:cm|ins:9|m:137438822400|ts:21.365811109542847
t:cm|ins:9|m:137438951424|ts:31.58132791519165
t:cm|ins:9|m:137438691328|ts:36.711552143096924
```

Figure 10. *Example of captured data*

3.3 Metrics

As explained in the previous chapter, Instant is a value that represents an index number on a hopping sequence. The value of the instant itself is not useful for this research, but the difference between instant and connection event counter is. Throughout this paper, the difference is referred to as Instant.
We are collecting Instant value statistics separately for all control PDUs that we are interested in, to test if there is any dependency on packet type.

When manually observing collected data - it became apparent that disabled channels are mostly grouped together, rather than scattered across the map spectrum. These groups are going to be called Ranges throughout the paper. It will be useful to have information about the number of ranges and their sizes to develop a smarter channel map recovery algorithm. Data about channel map update frequency will be useful for working on tapping into long-lived connections.

It will also be useful to collect data regarding used PHY modes and their frequency, as well as Window offset, Window size and interval values and frequency of their values, to be able to react to LL_CONNECTION_UPDATE_IND and LL_PHY_UPDATE_IND updates.

A custom script has been written in order to parse and visualize the data mentioned in the previous chapter. The code can be found on the following link: `https://github.com/stas-me/Sniffle/commit/2047e48eaf49c1abef3d3833a6c043c7a84f5f03`

3.4 Chart specifications

Channel map (Ex: Figure 13): Horizontal axis represents data-channel number and vertical axis represents number of times that the channel has been turned off during an update. Note: it also includes data collected from CONNECT_IND PDU, which represents the initial map, before it gets updated.

Map ranges (Ex: Figure 14): Range is a sequence of data channels that are turned off and are placed directly one after another, without an active channel placed in between. Channel map can have one or multiple ranges deactivated during an update. 0 ranges means that all 37 channels were set to be active.

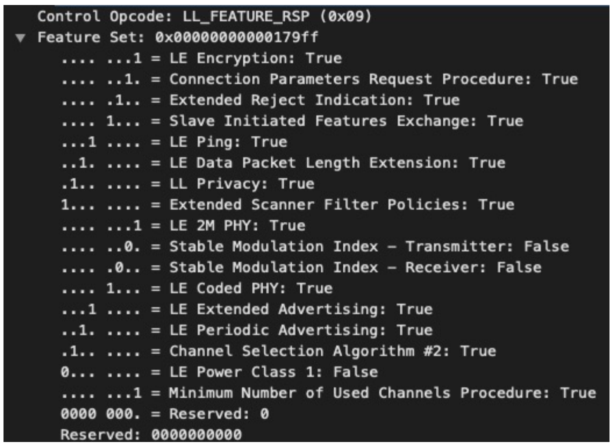
Time between channel map updates (Ex: Figure 15): Horizontal axis represents time in seconds. Time was measured with microsecond precision and then decimal part was dropped off, so for example the "2" bar includes every value in the range $2 \leq n < 3$. Grouped bars, like 14-17 represent values in range $14 \leq n < 17$

Instants (Ex: Figure 19): Horizontal axes represent instant value for corresponding PDU. The last chart displays combined data of previous charts.

Connection parameters (Ex: Figure 20): Horizontal axes represent corresponding connection or connection update parameter values.

3.5 iOS

All of the iOS tests were performed using iPhone 11 as a master and Xiaomi Mi 9T Pro as a slave. Both devices were using nRFConnect [37] to communicate with each other.
As we can see from Figure 11 - iPhone 11 supports Bluetooth v5 features like LE 2M PHY, LE Coded PHY, and CSA #2



```
    Control Opcode: LL_FEATURE_RSP (0x09)
▼   Feature Set: 0x00000000000179ff
        .... ...1 = LE Encryption: True
        .... ..1. = Connection Parameters Request Procedure: True
        .... .1.. = Extended Reject Indication: True
        .... 1... = Slave Initiated Features Exchange: True
        ...1 .... = LE Ping: True
        ..1. .... = LE Data Packet Length Extension: True
        .1.. .... = LL Privacy: True
        1... .... = Extended Scanner Filter Policies: True
        .... ...1 = LE 2M PHY: True
        .... ..0. = Stable Modulation Index - Transmitter: False
        .... .0.. = Stable Modulation Index - Receiver: False
        .... 1... = LE Coded PHY: True
        ...1 .... = LE Extended Advertising: True
        ..1. .... = LE Periodic Advertising: True
        .1.. .... = Channel Selection Algorithm #2: True
        0... .... = LE Power Class 1: False
        .... ...1 = Minimum Number of Used Channels Procedure: True
        0000 000. = Reserved: 0
        Reserved: 0000000000
```

Figure 11. *iPhone 11 features retrieved from LL_FEATURE_RSP packet*

3.5.1 By a router

Test description: both master and slave devices are placed within 0.5 metres from a router that is used for active and continuous downloading of a big file on the maximal available speed of 50 Mbit/s.
Both devices remain within the 0.5m range throughout the whole capturing process, including capturing the initial connection event.

Expected result: it is expected that close proximity to a device that actively emits interfering RF signal will significantly affect the transmission performance and force channel map to adjust in order to prevent the interference.

Test process: connection is established between two devices, and then they are lying still until enough data is collected. Figure 12 displays the Wi-Fi RF environment at the moment of performing the test. The environment was captured using NetSpot[1] application.

[1]NetSpot `https://play.google.com/store/apps/details?id=com.etwok.netspotapp`

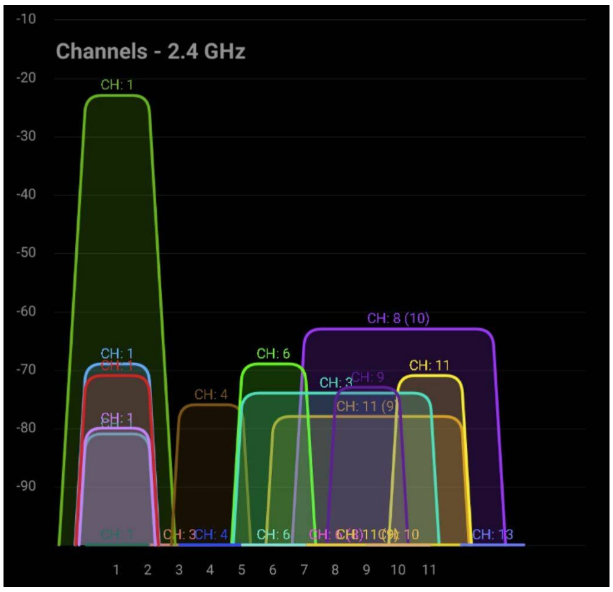


Figure 12. *Wi-Fi environment during the test captured with NetSpot*

Results:

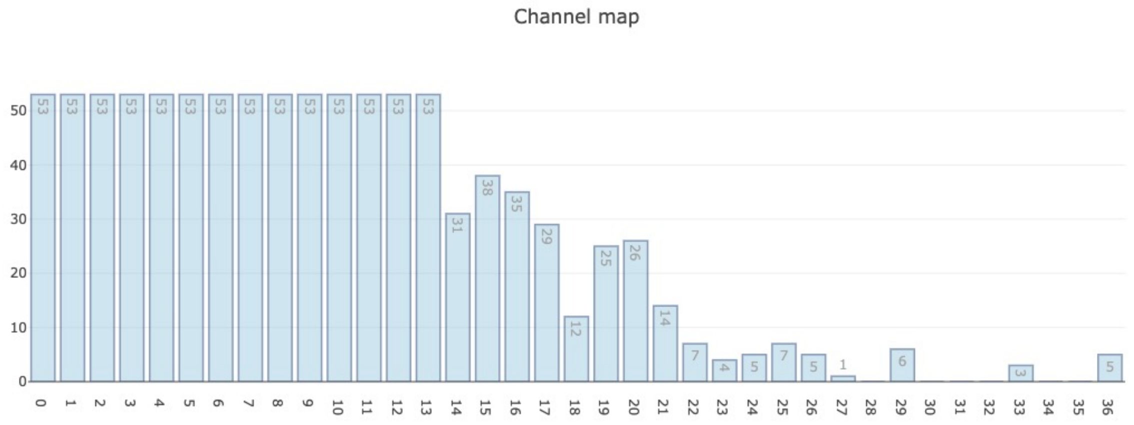


Figure 13. *iOS channel map by a router*

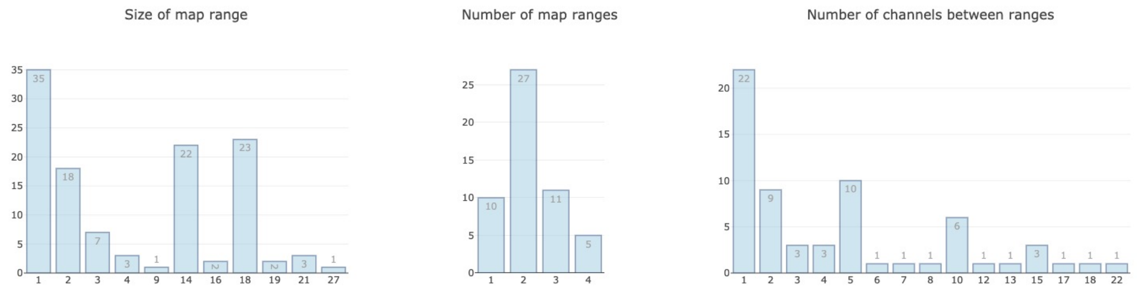


Figure 14. *iOS map ranges by a router*

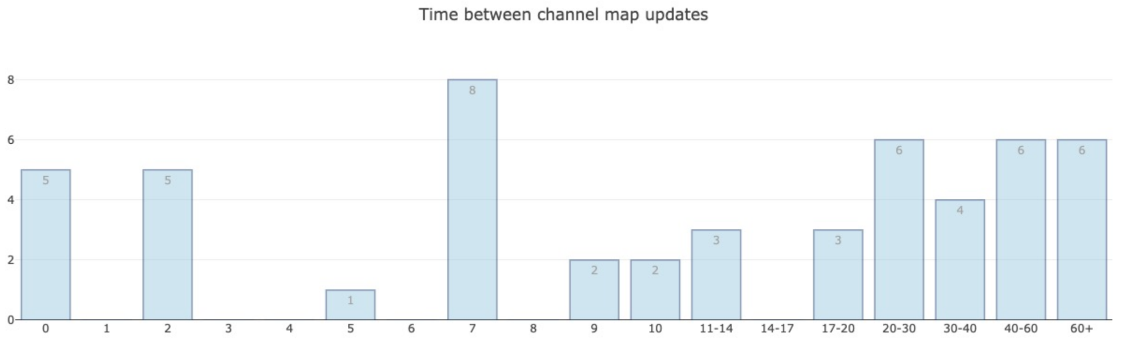


Figure 15. *iOS time between map updates by a router*

The test was running for 21 minutes and involved 52 channel map updates. As it is evident from Figure 13, channels 0-12 were turned off for all 52 updates and they were turned off during the connection event (in CONNECT_IND packet) as well. Wi-Fi channel 1 operates in 2401–2423 MHz frequency range, which correlates with 0-9 BLE data channels that operate in 2404-2422 MHz range (Figure 8). It is evident that a strong Wi-Fi signal in close proximity affected channel map updates. Much of the time, more than one range was switched off. Channel maps mostly looked like a big range of disabled channels closely followed by another smaller range.

3.5.2 Away from Wi-Fi networks

Test description: Both master and slave devices were brought to a place where the closest building was located 150 metres away (Figure 16).

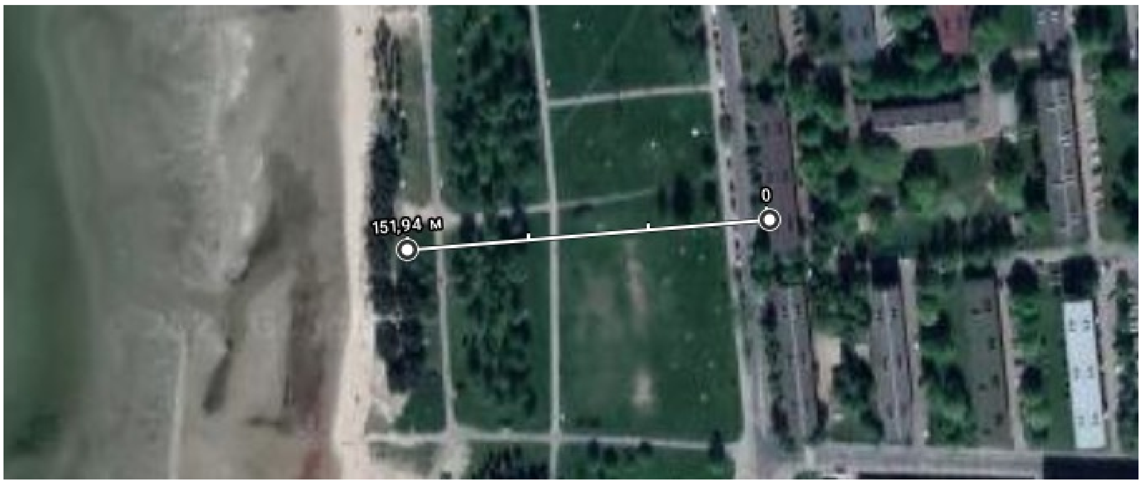


Figure 16. *Location of the performed test*

Expected result: it is expected that staying away from a big source of 2.4 GHz RF noise, that is house Wi-Fi access points, will positively affect the quality of communication between devices and therefore hopefully let devices communicate using all of the available channels.

Test process: connection is established between two devices, and then they are lying still, until enough data is collected. Most of the time, no Wi-Fi signal was being captured, but occasionally a channel with minimal RSSI would appear on the map and then soon disappear (Figure 17).

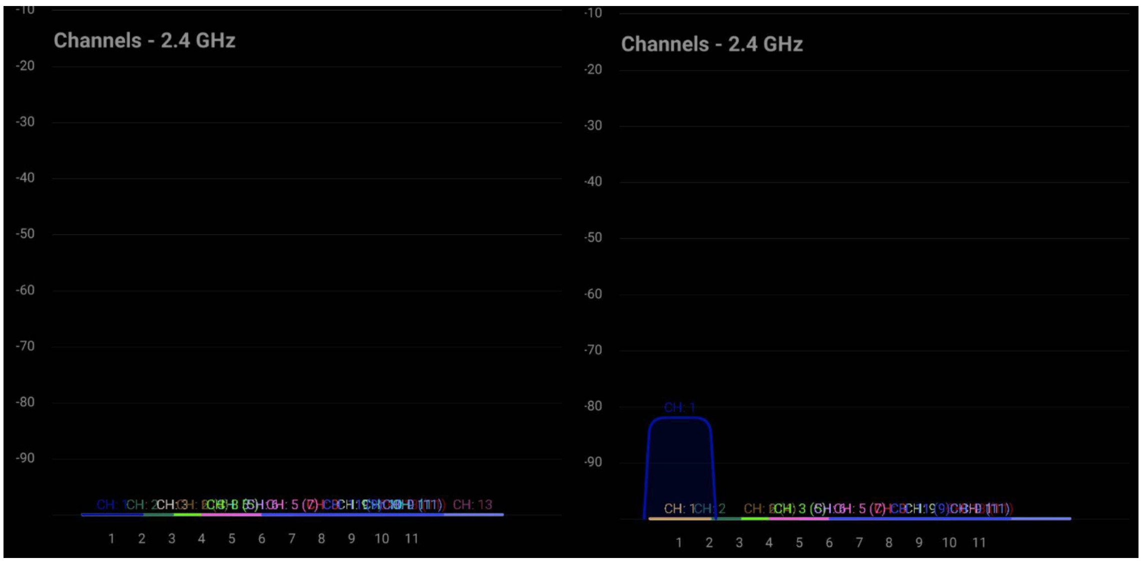


Figure 17. *Wi-Fi environment during the test*

Results:

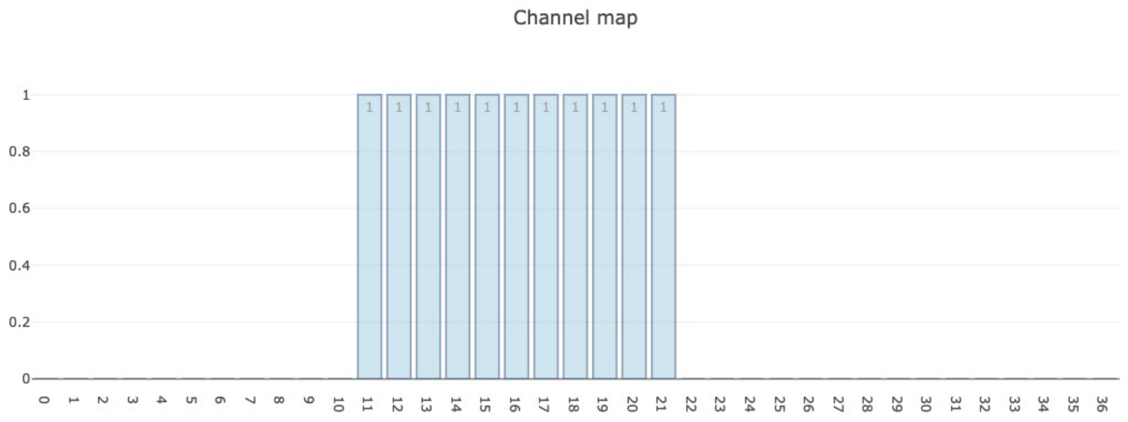


Figure 18. *iOS channel map away from Wi-Fi networks*

The test was running for 13 minutes. The connection was established with all 37 data channels enabled. The channel map update from Figure 18 happened 30 seconds before the end of the test, and author failed to notice it on the spot, so Wi-Fi noise at that particular moment is unknown, but the communication was performed with all of the channels enabled for most of the time.

3.5.3 Combined data

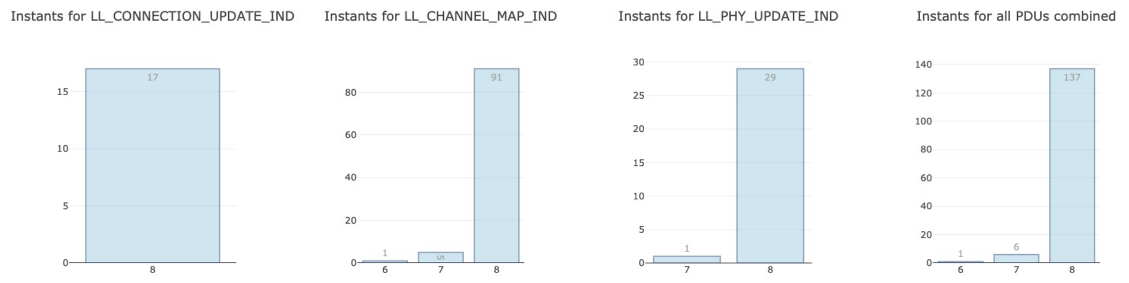


Figure 19. *iOS instants combined*

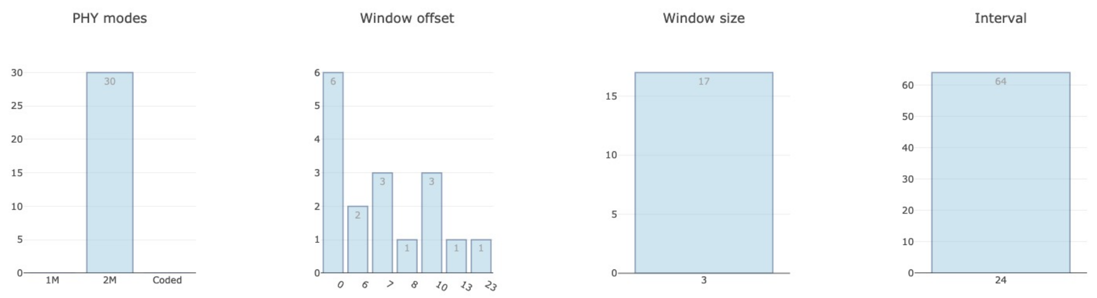


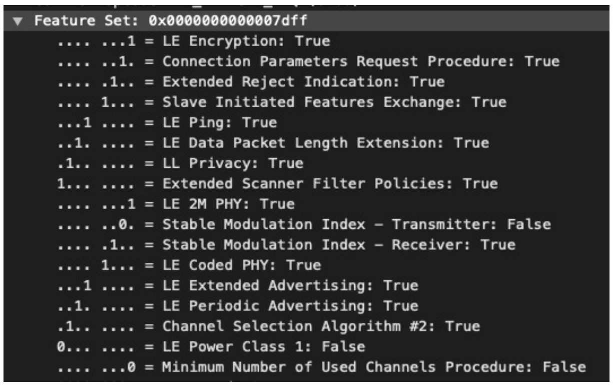
Figure 20. *iOS connection parameters combined*

The data was retrieved from a total of around 88 minutes of captured traffic that involved 47 connection initiations. Both the most frequent and the biggest instant was 8. 2M PHY was turned on within seconds from the connection initiation and has never been observed to be switched back to 1M PHY or to Coded PHY. Window offset varies a lot in size, but Window size was always equal to 3, and Interval was never observed to be different from 24.

Several attempts were made to affect Interval value, but none were successful. nRFConnect's Macros functionality was used to repetitively send 18 byte long strings with maximal frequency. 18 byte was the maximal length of data fitting in a single PDU, and anything bigger than that was being split into segments. The frequency ended up being around 12 packets per second. In one test, macros was turned on for 3 minutes and in the other one for 10 minutes, but no Interval change has occurred.

3.6 Android

All of the Android tests were performed using Xiaomi Mi 9T Pro as a master and OnePlus 8 Pro as a slave. Both devices were using nRFConnect [37] to communicate with each other.
As we can see from Figure 21, Xiaomi Mi 9T Pro supports Bluetooth v5 features like LE 2M PHY, LE Coded PHY, and CSA #2



```
▼ Feature Set: 0x0000000000007dff
    .... ...1 = LE Encryption: True
    .... ..1. = Connection Parameters Request Procedure: True
    .... .1.. = Extended Reject Indication: True
    .... 1... = Slave Initiated Features Exchange: True
    ...1 .... = LE Ping: True
    ..1. .... = LE Data Packet Length Extension: True
    .1.. .... = LL Privacy: True
    1... .... = Extended Scanner Filter Policies: True
    .... ...1 = LE 2M PHY: True
    .... ..0. = Stable Modulation Index - Transmitter: False
    .... .1.. = Stable Modulation Index - Receiver: True
    .... 1... = LE Coded PHY: True
    ...1 .... = LE Extended Advertising: True
    ..1. .... = LE Periodic Advertising: True
    .1.. .... = Channel Selection Algorithm #2: True
    0... .... = LE Power Class 1: False
    .... ...0 = Minimum Number of Used Channels Procedure: False
```

Figure 21. *Xiaomi Mi 9T Pro features retrieved from LL_FEATURE_RSP packet*

3.6.1 By a router

The same test as described in Chapter 3.5.1 was performed for Android, so the test description, expected result, and test process are the same. The Wi-Fi environment during the test is presented on Figure 22

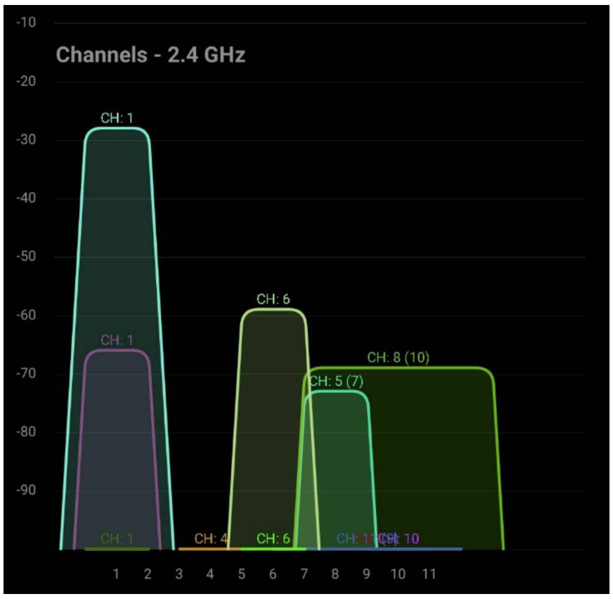


Figure 22. *Wi-Fi environment during the test captured with NetSpot*

Results:

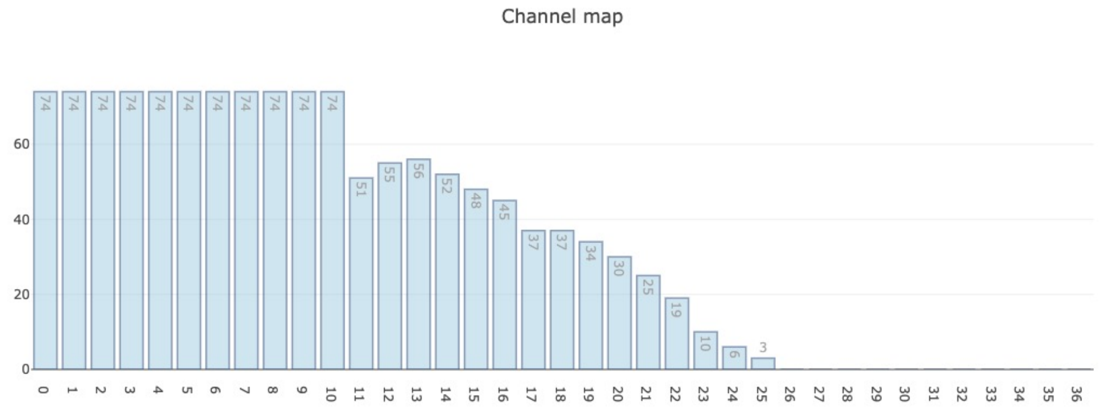


Figure 23. *Android channel map by a router*

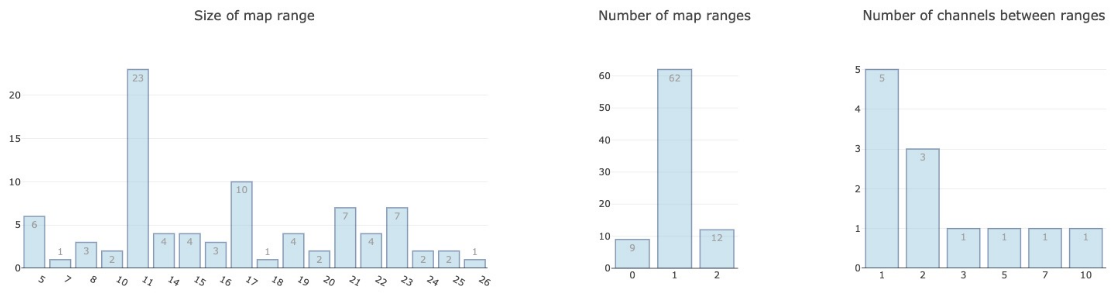


Figure 24. *Android map ranges by a router*

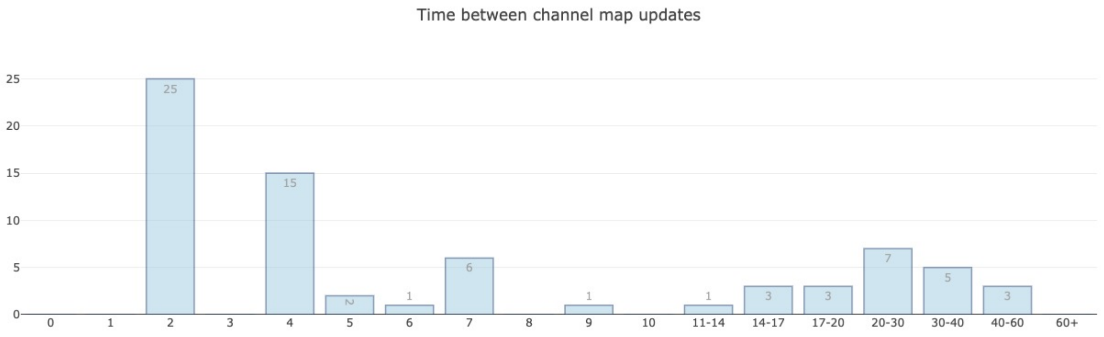


Figure 25. *Android time between map updates by a router*

15 minutes of communication was captured for the test, and it involved 74 map updates. During the test, connection was established 9 times, and all of the times, unlike iOS (Chapter 3.5.1), communication started with all channels activated.
Capturing data for this particular test was rather problematic. OnePlus 8 Pro was observed to have noticeably lower RSSI compared to Xiaomi Mi 9T Pro. Generally, in less noisy conditions, when both devices were located within 20 cm from the CC2652R1 board,

Xiaomi had RSSI between -30 and -40, while OnePlus was rarely going above -60. CC2652R1 had trouble capturing advertisement packets on channel 37, since it is located at the very beginning of the 40 channel spectrum (Figure 2). Packets were getting lost, and it negatively affected Sniffle's ability to follow advertisements on channels 38 and 39, since the functionality is dependent on packet reception time measurement. In addition to that, whenever connection initiation happened, and Sniffle was able to capture it, the first map update was happening only after about 200 hops. Before it was getting updated - the full channel spectrum was used for communication. Packets were continuously getting lost, and even though Sniffle has slave latency parameter internally increased by 3, to tolerate occasionally missed packets, it was not enough, and connection was getting considered lost. On one occasion, it took 7 successfully captured connection initiations in a row to get one that would live through 200 hops until the first channel map update.
Close proximity to the router affected only CC2652R1's ability to capture packets, though. Xiaomi and OnePlus were connecting and operating without any noticeable issue.
As it is evident from Figure 23, channels 0-10 were turned off for all 74 updates. Wi-Fi channel 1 operates in 2401–2423 MHz frequency range, which correlates with 0-9 BLE data channels that operate in 2404-2422 MHz range (Figure 8). It is evident that a strong Wi-Fi signal in close proximity affected channel map updates. Much of the time, only one range was switched off. Channel maps mostly looked like a big range of disabled channels located at the beginning of the spectrum. Sometimes it was closely followed by another range that was smaller in size.
55% of the updates happened within 5 seconds from the preceding update.

3.6.2 Away from Wi-Fi networks

The same test as described in Chapter 3.5.2 was performed for Android, so the test description, expected result, test process and the test environment are the same. Tests were performed one after another in the same time window.

Result: Test was running for 13 minutes. Connection was established with all 37 data channels enabled, and channel update has not happened during the experiment.

3.6.3 Combined data

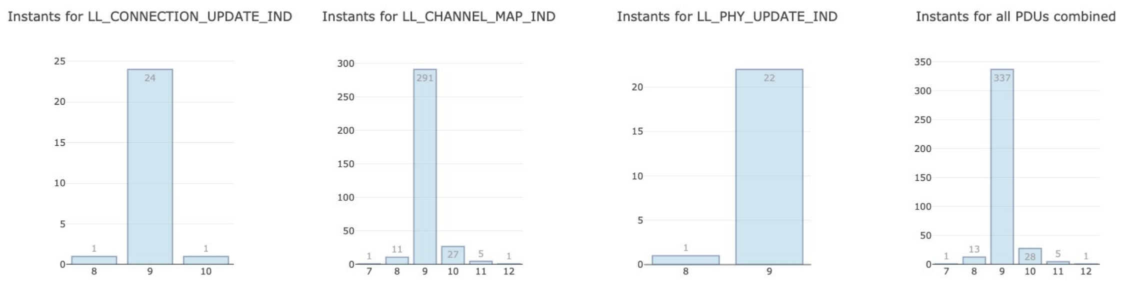


Figure 26. *Android instants combined*

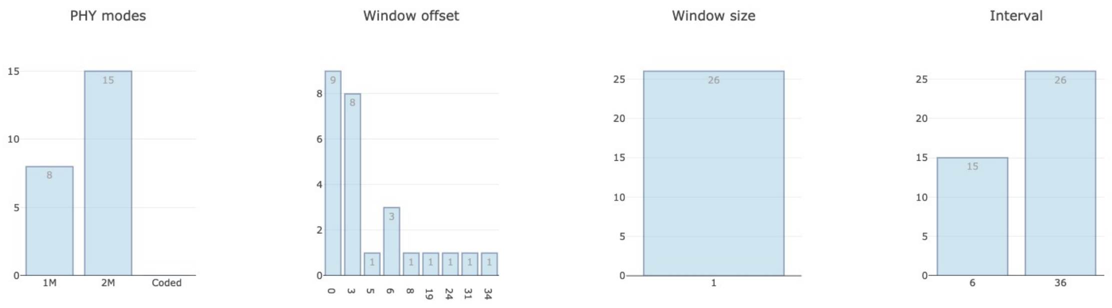


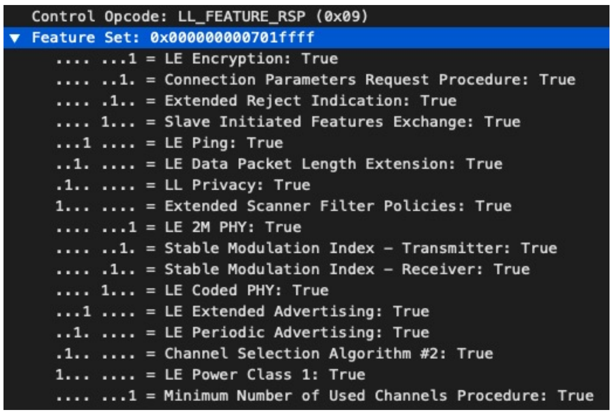
Figure 27. *Android connection parameters combined*

The data was retrieved from a total of around 187 minutes of captured traffic that involved 16 connection initiations. Instants were in the range from 7 to 12, and the most frequent instant was 9. It was observed in 87.5% of cases. Instants 8 and 10 are 10.9% in total, and the 1.6% was for other values.
Android, unlike iOS (Chapter 3.5.3), was not switching to 2M PHY right away. Cases have been observed, where communication was switched to 2M PHY only after 5 minutes from the connection initiation. It has also been observed to be switched back to 1M PHY. Coded PHY has never been used. Window offset varies a lot in size, but Window size was always equal to 1, and Interval was never observed to be different from 36 or 6.
The same test as described in Chapter 3.5.3 was performed for Android to try to affect interval value and it did not have any effect.
Connection was always initialized with an Interval value of 36. Then, within 0.5 - 3 seconds Interval changed to 6, the devices exchanged GATT characteristics in 5-10 seconds, and then Interval was updated back to 36 and did not change afterwards.

3.7 Android - Gabeldorsche

Gabeldorsche is a Bluetooth stack for Android developed by Google. As of April 2021, it is under development, and Google has not announced its release date, or version of Android that will have Gabeldorsche as the main stack yet, but it is already available for some devices running Android 11 and can be turned on in developer options.

All of the Gabeldorsche tests were performed using OnePlus 8 Pro as a master and Xiaomi Mi 9T Pro as a slave. Both devices were using nRFConnect [37] to communicate with each other.
As we can see from Figure 28, OnePlus 8 Pro supports all of the Bluetooth v5 features.



```
  Control Opcode: LL_FEATURE_RSP (0x09)
▼ Feature Set: 0x000000000701ffff
      .... ...1 = LE Encryption: True
      .... ..1. = Connection Parameters Request Procedure: True
      .... .1.. = Extended Reject Indication: True
      .... 1... = Slave Initiated Features Exchange: True
      ...1 .... = LE Ping: True
      ..1. .... = LE Data Packet Length Extension: True
      .1.. .... = LL Privacy: True
      1... .... = Extended Scanner Filter Policies: True
      .... ...1 = LE 2M PHY: True
      .... ..1. = Stable Modulation Index - Transmitter: True
      .... .1.. = Stable Modulation Index - Receiver: True
      .... 1... = LE Coded PHY: True
      ...1 .... = LE Extended Advertising: True
      ..1. .... = LE Periodic Advertising: True
      .1.. .... = Channel Selection Algorithm #2: True
      1... .... = LE Power Class 1: True
      .... ...1 = Minimum Number of Used Channels Procedure: True
```

Figure 28. *OnePlus 8 Pro with enabled Gabeldorsche stack features retrieved from LL_FEATURE_RSP packet*

3.7.1 By a router

The same test as described in Chapter 3.5.1 was performed for Gabeldorsche, so the test description, expected result, and test process are the same. The Wi-Fi environment during the test is presented on Figure 29

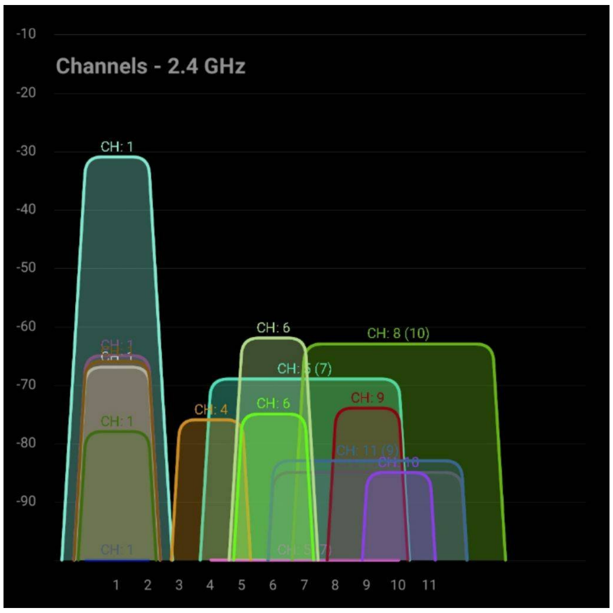


Figure 29. *Wi-Fi environment during the test captured with NetSpot*

Results:

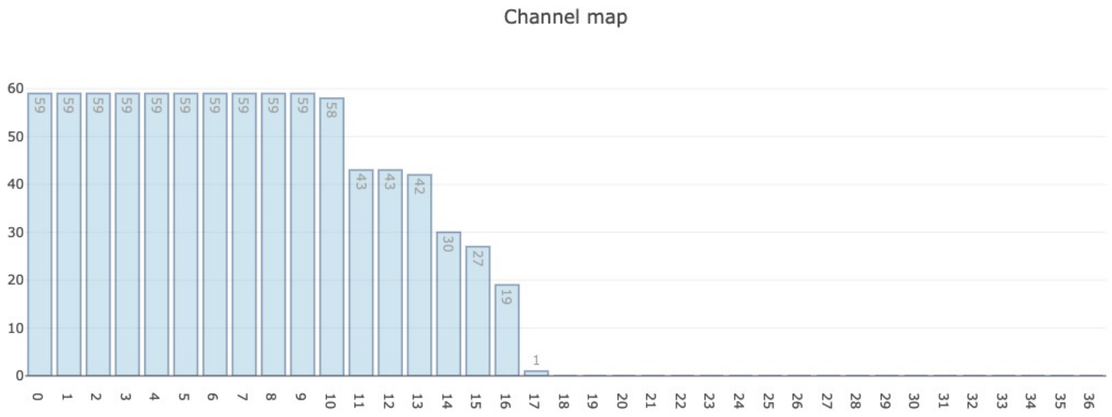


Figure 30. *Gabeldorsche channel map by a router*

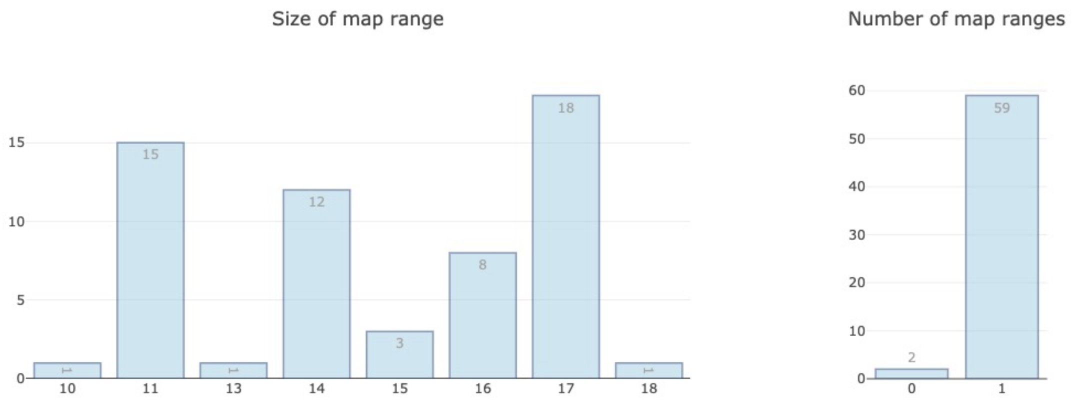


Figure 31. *Gabeldorsche map ranges by a router*

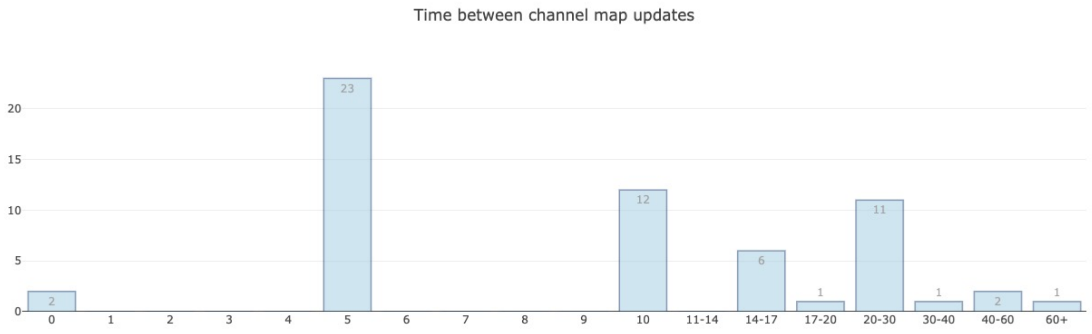


Figure 32. *Gabeldorsche time between map updates by a router*

13 minutes of communication was captured for the test, and it involved 60 map updates. Unlike iOS (Chapter 3.5.1), communication started with all channels activated, as it was with Android (Chapter 3.6.1) and Mac (Chapter 3.8.1).
Unlike Android, iOS, and Mac, it did not have these channels turned off during the whole connection. There was a period between 203 and 228 seconds of the connection, where all of the channels were enabled. This behavior is unexpected, but can probably be neglected due to the stack still being under development. It happened only 1 time out of 60.
As it is evident from Figure 30, channels 0-9 were turned off for 59 updates out of 60, and channel 10 was turned off 58 times. Wi-Fi channel 1 operates in 2401–2423 MHz frequency range, which correlates with 0-9 BLE data channels that operate in 2404-2440 MHz range (Figure 8). It is evident that a strong Wi-Fi signal in close proximity affected channel map updates.
Unlike other platforms, Gabeldorsche only had 1 map range at a time.

As we can see from Figure 32 - 38% of the updates happened within 6 seconds from the preceding update. The general pattern of the changes was as follows: when the range was towards its biggest size, it tried to narrow it down and then in 5-6 seconds made it bigger again.

3.7.2 Away from Wi-Fi networks

The same test as described in Chapter 3.5.2 was performed for Gabeldorsche, so the test description, expected result, test process, and the test environment are the same. Tests were performed one after another in the same time window.

Result: Test was running for 11 minutes. Connection was established with all 37 data channels enabled, and channel update has not happened during the experiment.

3.7.3 Combined data

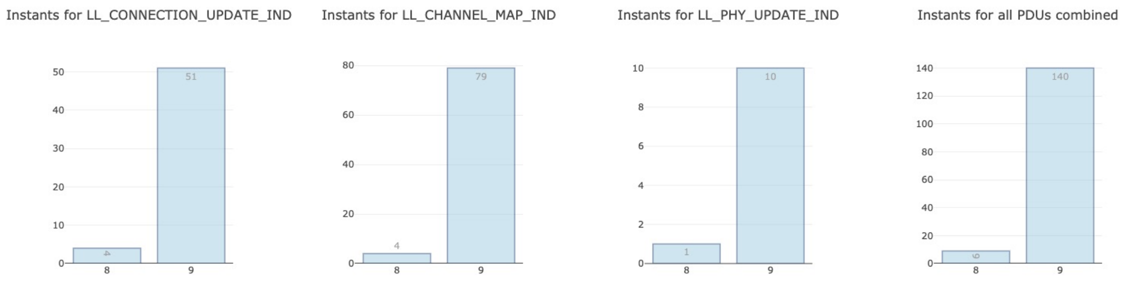


Figure 33. *Gabeldorsche instants combined*

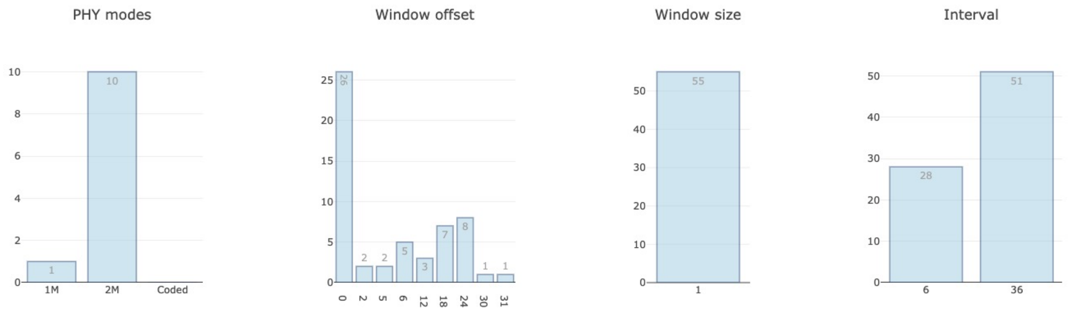


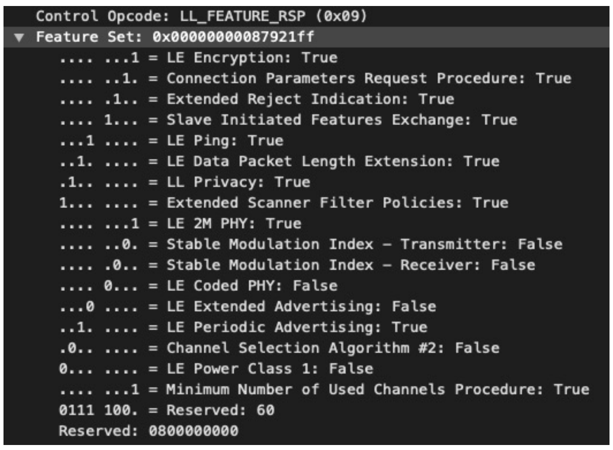
Figure 34. *Gabeldorsche connection parameters combined*

The data was retrieved from a total of around 45 minutes of captured traffic that involved 24 connection initiations. 94% of instants were equal to 9, and 6% was equal to 8. No other value has been observed. Gabeldorsche's pool of available instants seems to be much smaller than Android's (Chapter 3.6.3)
Gabeldorsche uses PHY modes exactly like Android does: does not switch to 2M PHY

right away, never used Coded PHY, has switched back to 1M PHY.
Window offset varies a lot in size, but Window size was always equal to 1, and Interval was never observed to be different from 36 or 6.
The same test as described in Chapter 3.5.3 was performed for Gabeldorsche to try to affect interval value, and it did not have any effect.
Like for Android - connection was always initialized with an Interval value of 36. Then, within 0.5 - 3 seconds Interval changed to 6, the devices exchanged GATT characteristics in 5-10 seconds, and then Interval was updated back to 36 and did not change afterwards.
Gabeldorsche was never observed to have more than 1 channel map range disabled.

3.8 Mac

All of the Mac tests were performed using Macbook Air 2020, which is advertised to have Bluetooth 5.0 support, as a master and Xiaomi Mi 9T Pro as a slave. Xiaomi was using nRFConnect [37], and Macbook was using BlueSee [38] to communicate with each other.
As we can see from Figure 35, Macbook Air 2020 does **not** support Bluetooth v5 features like LE Coded PHY and CSA #2, and as it becomes evident from the performed tests, it does not use LE 2M PHY.



```
  Control Opcode: LL_FEATURE_RSP (0x09)
▼ Feature Set: 0x00000000087921ff
    .... ...1 = LE Encryption: True
    .... ..1. = Connection Parameters Request Procedure: True
    .... .1.. = Extended Reject Indication: True
    .... 1... = Slave Initiated Features Exchange: True
    ...1 .... = LE Ping: True
    ..1. .... = LE Data Packet Length Extension: True
    .1.. .... = LL Privacy: True
    1... .... = Extended Scanner Filter Policies: True
    .... ...1 = LE 2M PHY: True
    .... ..0. = Stable Modulation Index - Transmitter: False
    .... .0.. = Stable Modulation Index - Receiver: False
    .... 0... = LE Coded PHY: False
    ...0 .... = LE Extended Advertising: False
    ..1. .... = LE Periodic Advertising: True
    .0.. .... = Channel Selection Algorithm #2: False
    0... .... = LE Power Class 1: False
    .... ...1 = Minimum Number of Used Channels Procedure: True
    0111 100. = Reserved: 60
    Reserved: 0800000000
```

Figure 35. *Macbook Air 2020 features retrieved from LL_FEATURE_RSP packet*

3.8.1 By a router

The same test as described in Chapter 3.5.1 was performed for Mac, so the test description, expected result, and test process are the same. The Wi-Fi environment during the test is presented on Figure 36.

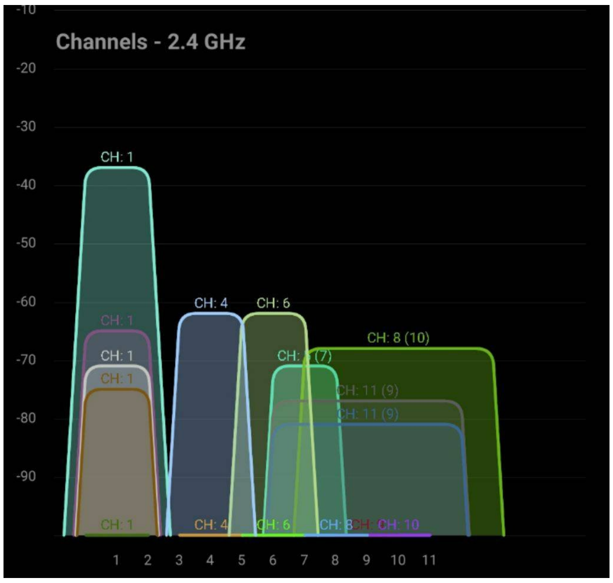


Figure 36. *Wi-Fi environment during the test captured with NetSpot*

Results:

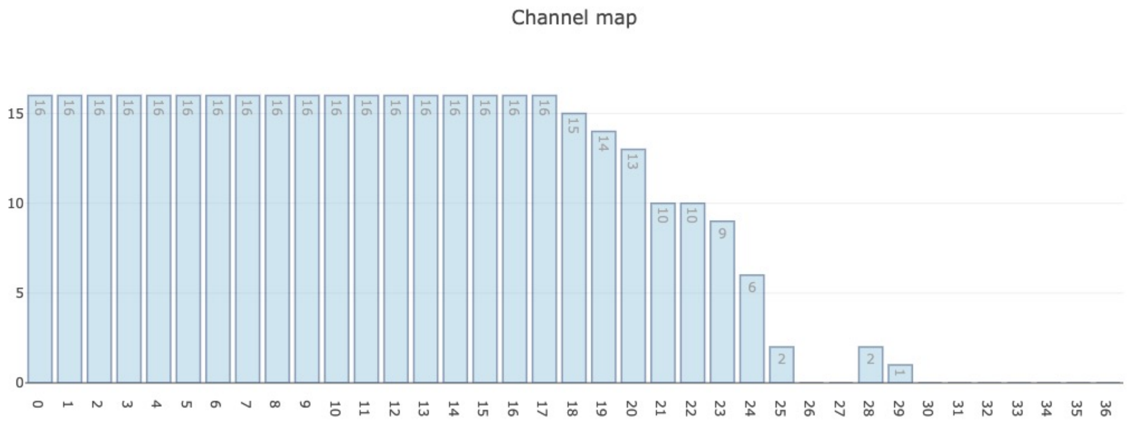


Figure 37. *Mac channel map by a router*

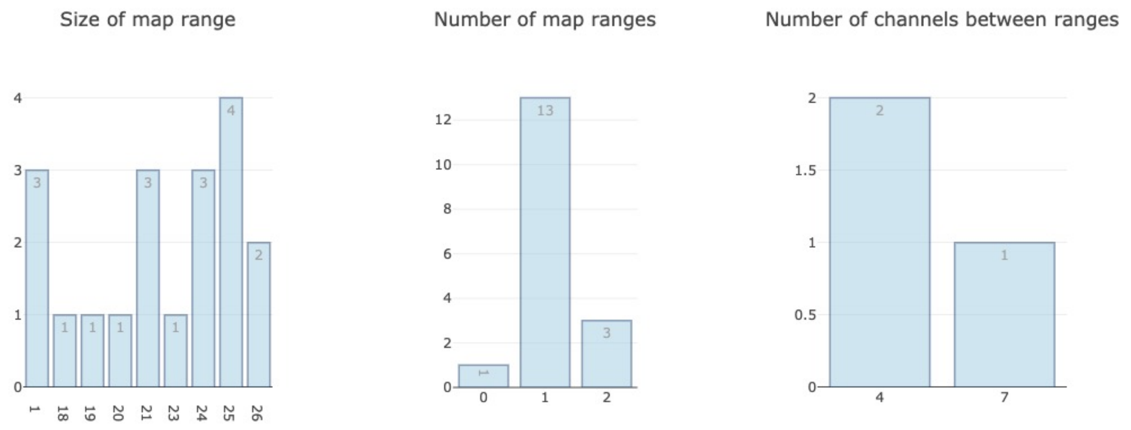


Figure 38. *Mac map ranges by a router*

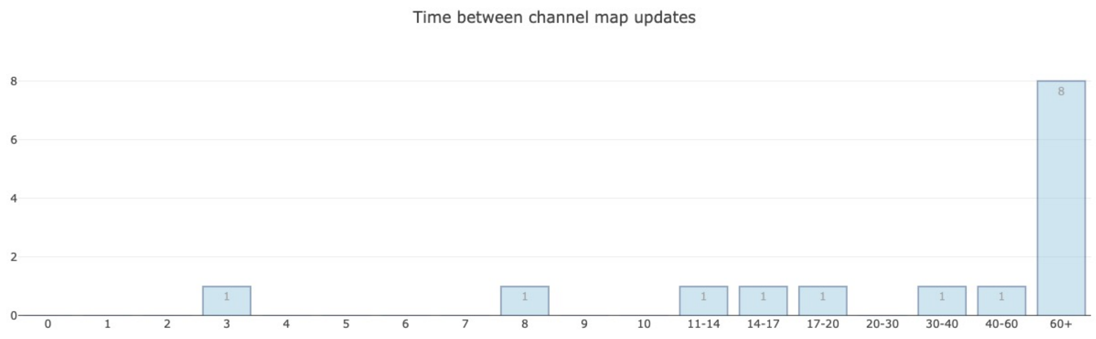


Figure 39. *Mac time between map updates by a router*

22 minutes of communication was captured for the test, and it involved 16 map updates. Unlike iOS (Chapter 3.5.1), communication started with all channels activated, as it was with Android (Chapter 3.6.1) and Gabeldorsche (Chapter 3.7.1).
Wi-Fi channels 1-5 operate in 2401–2443 MHz frequency range, which correlates with 0-17 BLE data channels that operate in 2404-2440 MHz range (Figure 8). It is evident that a strong Wi-Fi signal in close proximity affected channel map updates.
Most of the time, only one range was disabled, and the 3 times when there were 2 ranges, it was 1 big range + 1 channel (28 or 29)

3.8.2 Away from Wi-Fi networks

The same test as described in Chapter 3.5.2 was performed for Android, so the test description, expected result, test process, and the test environment are the same. Tests were performed one after another in the same time window.

Result:

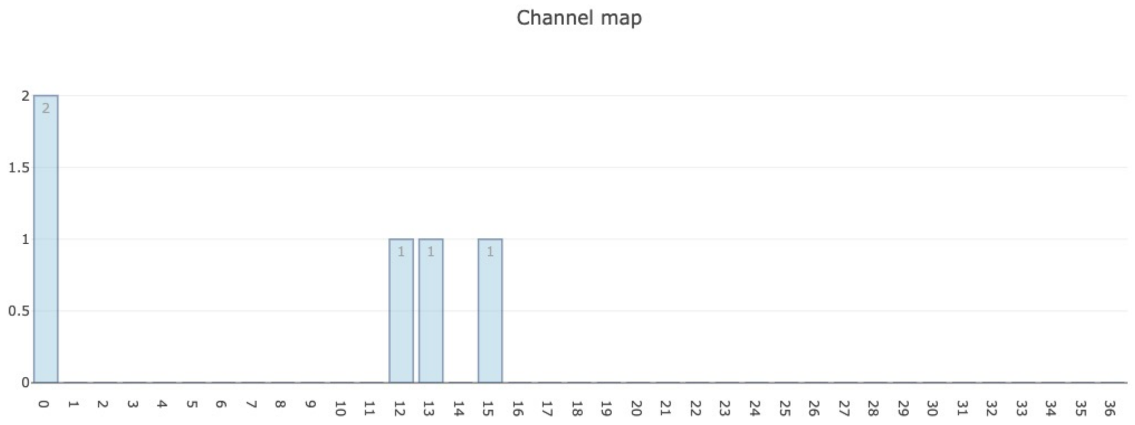


Figure 40. *Mac channel map away from Wi-Fi networks*

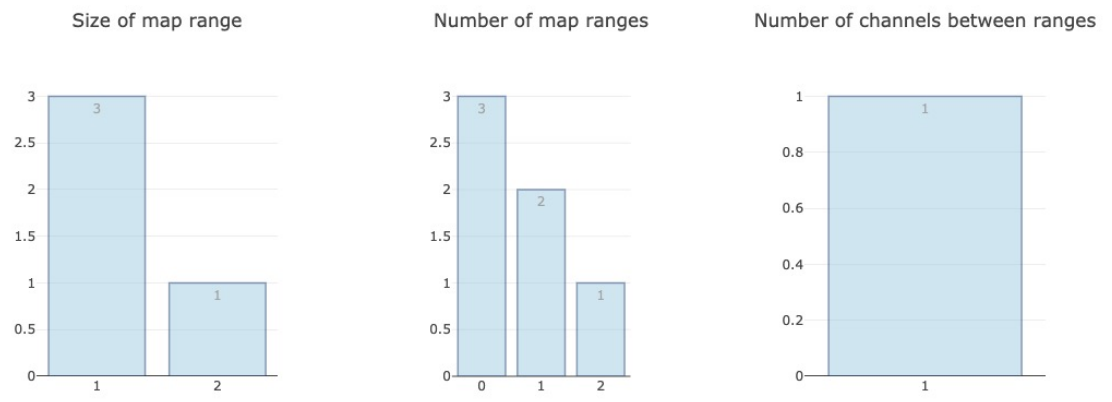


Figure 41. *Mac channel map ranges away from Wi-Fi networks*

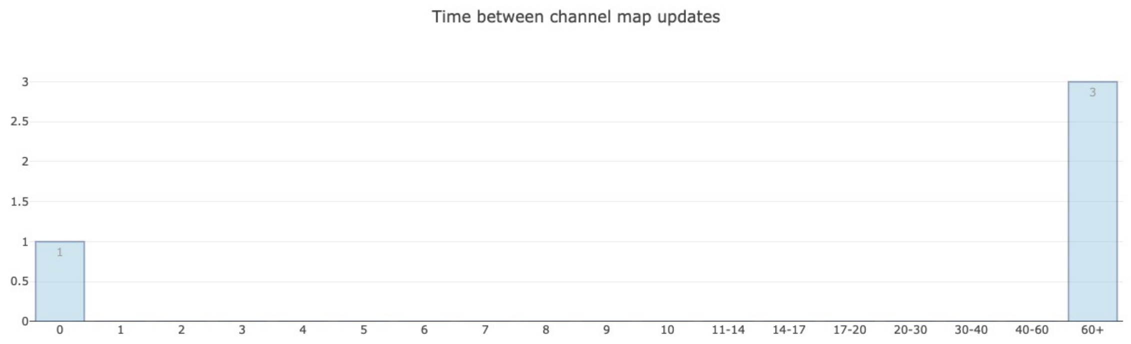


Figure 42. *Mac time between map updates away from Wi-Fi networks*

The test was running for 12 minutes. Connection was established with all 37 data channels enabled. 4 channel updates happened during the experiment, disabling 1-3 channels at a time.

3.8.3 Combined data

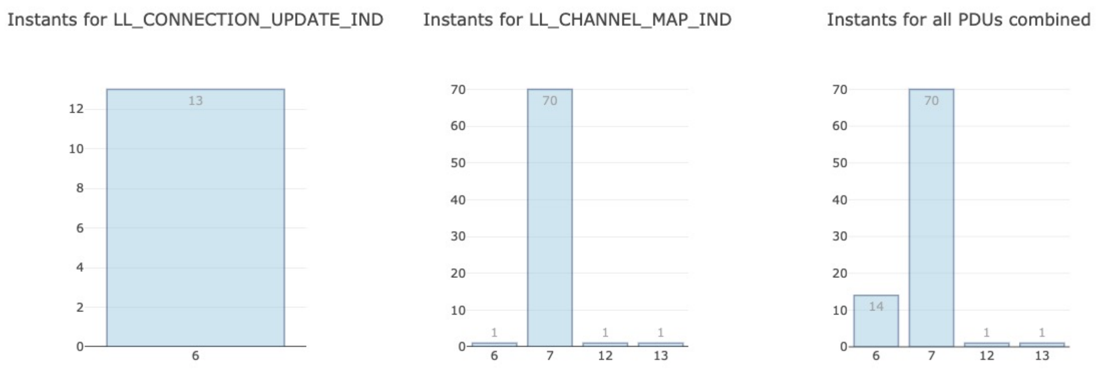


Figure 43. *Mac instants combined*

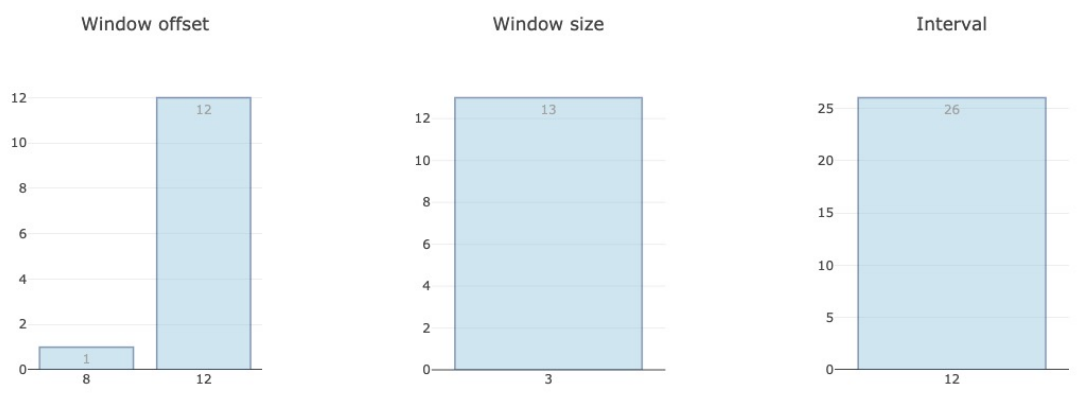
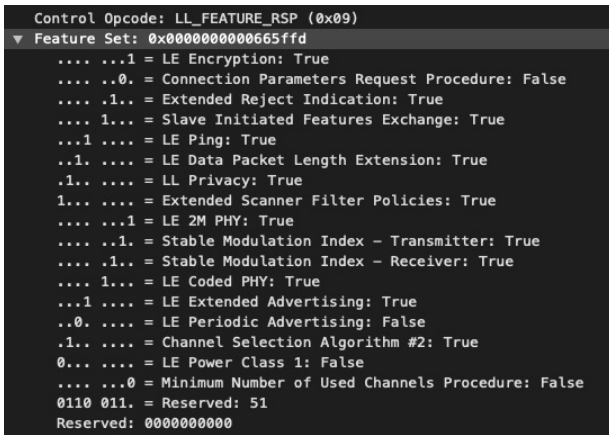


Figure 44. *Mac connection parameters combined*

The data was retrieved from a total of around 46 minutes of captured traffic that involved 13 connection initiations.
81% of instants were equal to 7, 16% was equal to 6. Instant values of 12 and 13 were observed 1 time each.
Interval was never observed to be different from 12.
2M PHY was never observed to be used.
Window offset is mostly 12, and Window size was always equal to 3, and Interval was never observed to be different from 12.
The same test as described in Chapter 3.5.3 was performed for Mac to try to affect interval value, and it did not have any effect.

3.9 Linux

All of the Linux tests were performed using ASUS USB-BT500 dongle on Ubuntu 21 with BlueZ Bluetooth stack as a master and Xiaomi Mi 9T Pro as a slave. Xiaomi was using nRFConnect [37], and Linux was using bluetoothctl utility from bluez-utils pack to communicate with each other. As we can see from Figure 45, ASUS USB-BT500 + BlueZ support Bluetooth v5 features like LE 2M PHY, LE Coded PHY, and CSA #2



```
  Control Opcode: LL_FEATURE_RSP (0x09)
▼ Feature Set: 0x0000000000665ffd
    .... ...1 = LE Encryption: True
    .... ..0. = Connection Parameters Request Procedure: False
    .... .1.. = Extended Reject Indication: True
    .... 1... = Slave Initiated Features Exchange: True
    ...1 .... = LE Ping: True
    ..1. .... = LE Data Packet Length Extension: True
    .1.. .... = LL Privacy: True
    1... .... = Extended Scanner Filter Policies: True
    .... ...1 = LE 2M PHY: True
    .... ..1. = Stable Modulation Index - Transmitter: True
    .... .1.. = Stable Modulation Index - Receiver: True
    .... 1... = LE Coded PHY: True
    ...1 .... = LE Extended Advertising: True
    ..0. .... = LE Periodic Advertising: False
    .1.. .... = Channel Selection Algorithm #2: True
    0... .... = LE Power Class 1: False
    .... ...0 = Minimum Number of Used Channels Procedure: False
    0110 011. = Reserved: 51
    Reserved: 0000000000
```

Figure 45. *ASUS USB-BT500 features retrieved from LL_FEATURE_RSP packet on linux*

Author could not organize "By a router" and "Away from Wi-Fi networks" experiments for Linux, so data was collected in a residential building environment. The goal is to have a look at other parameters. The Wi-Fi environment during the test is presented on Figure 46

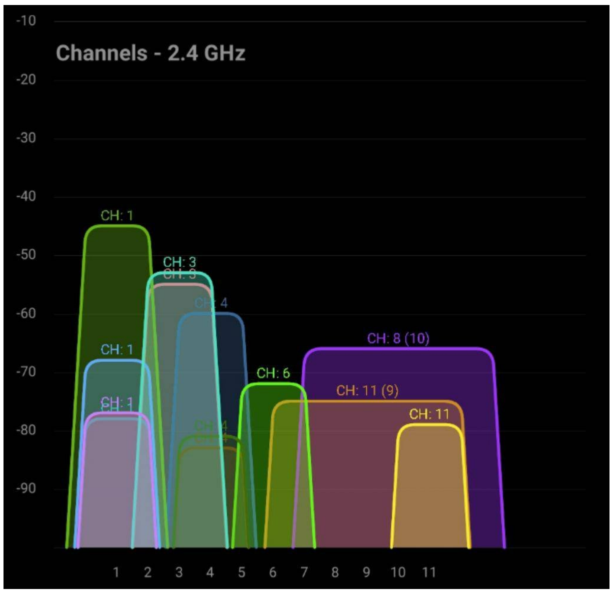


Figure 46. *Wi-Fi environment during the test captured with NetSpot*

Results:

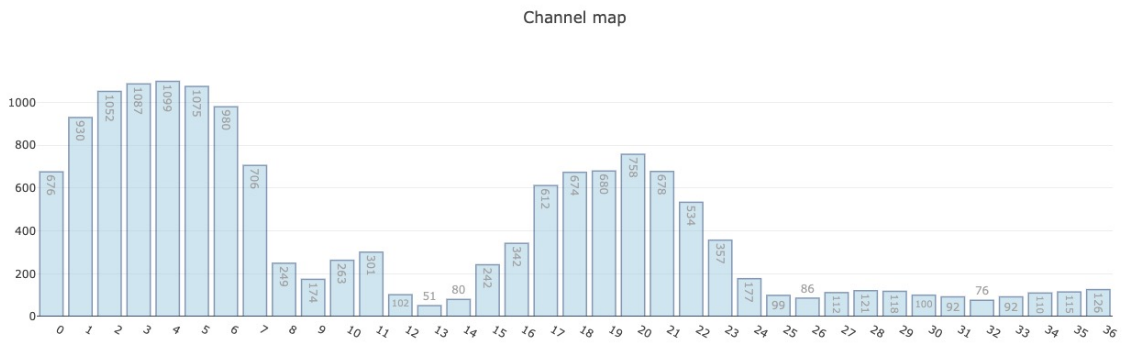


Figure 47. *Linux channel map by a router*

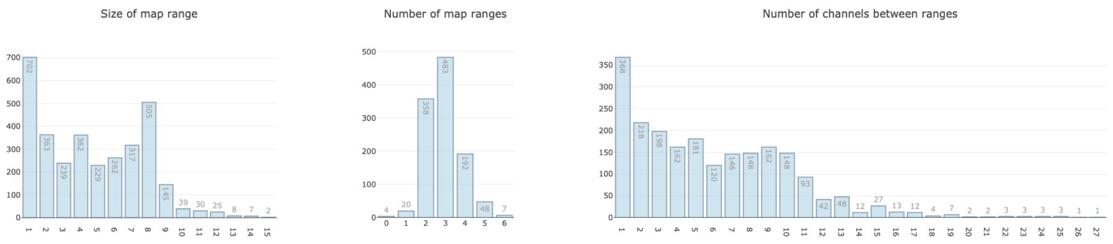


Figure 48. *Linux map ranges by a router*

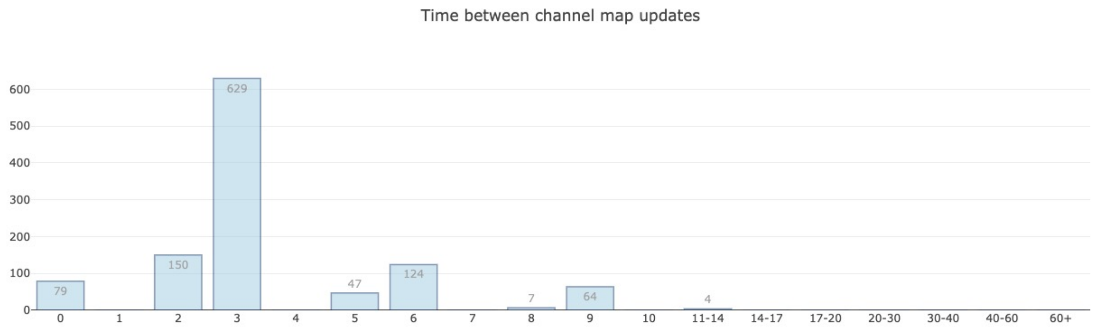


Figure 49. *Linux time between map updates by a router*

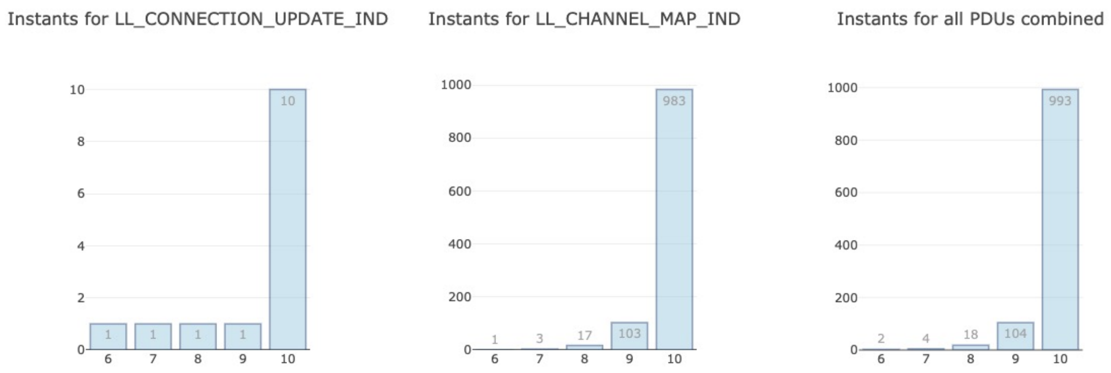


Figure 50. *Linux instants combined*

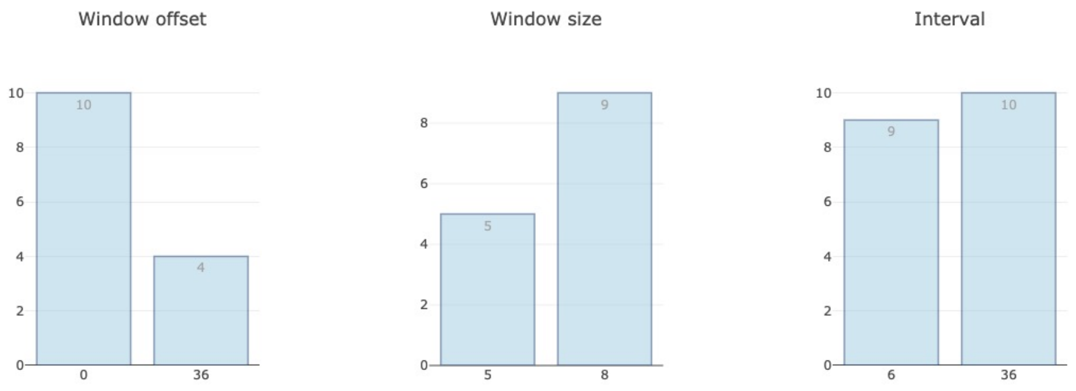
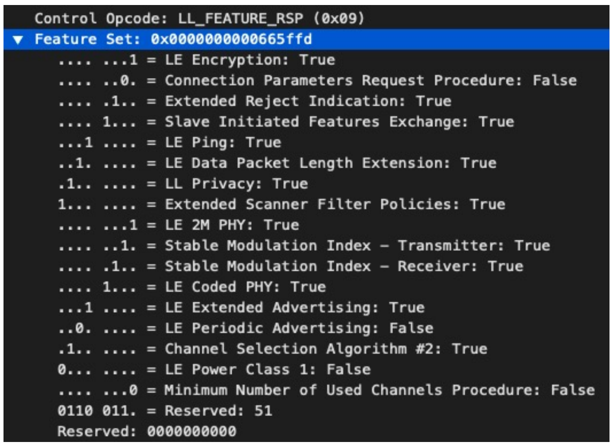


Figure 51. *Linux connection parameters combined*

The data was retrieved from a total of around 68 minutes of captured traffic that involved 5 connection initiations. Both the most frequent and the biggest instant was 10. It was used 88.6% of the time. The second most frequent was instant 9 with 9.3%, and the rest was instant 6, 7, and 8.
Neither 2M, nor Coded PHYs have ever been observed to be used.
Window offset in 71% cases was equal to 0 and in 39% equal to 36. 2 different values were observed for Window size: 5 and 8, and Interval was never observed to be different from 36 or 6.
The same test as described in Chapter 3.5.3 was performed for Linux to try to affect interval value, and it did not have any effect.
Like for Android (Chapter 3.6.3) - connection was always initialized with an Interval value of 36. Then, within 0.5 - 3 seconds Interval changed to 6, the devices exchanged GATT characteristics in 5-10 seconds, and then Interval was updated back to 36 and did not change afterwards.
One of the 5 connection initiations was noticed to have 2 ranges disabled on the channel map.

3.10 Windows

All of the Linux tests were performed using ASUS USB-BT500 dongle on Windows 10 as a master and Xiaomi Mi 9T Pro as a slave. Xiaomi was using nRFConnect [37], and Windows' native GUI interface was used to discover and connect to Xiaomi device. As we can see from Figure 45, ASUS USB-BT500 + Windows support the same list of Bluetooth v5 features as USB-BT500 + BlueZ, like LE 2M PHY, LE Coded PHY, and CSA #2



```
  Control Opcode: LL_FEATURE_RSP (0x09)
▼ Feature Set: 0x0000000000665ffd
     .... ...1 = LE Encryption: True
     .... ..0. = Connection Parameters Request Procedure: False
     .... .1.. = Extended Reject Indication: True
     .... 1... = Slave Initiated Features Exchange: True
     ...1 .... = LE Ping: True
     ..1. .... = LE Data Packet Length Extension: True
     .1.. .... = LL Privacy: True
     1... .... = Extended Scanner Filter Policies: True
     .... ...1 = LE 2M PHY: True
     .... ..1. = Stable Modulation Index - Transmitter: True
     .... .1.. = Stable Modulation Index - Receiver: True
     .... 1... = LE Coded PHY: True
     ...1 .... = LE Extended Advertising: True
     ..0. .... = LE Periodic Advertising: False
     .1.. .... = Channel Selection Algorithm #2: True
     0... .... = LE Power Class 1: False
     .... ...0 = Minimum Number of Used Channels Procedure: False
     0110 011. = Reserved: 51
     Reserved: 0000000000
```

Figure 52. *ASUS USB-BT500 features retrieved from LL_FEATURE_RSP packet on Windows*

Author was not able to organize "By a router" and "Away from Wi-Fi networks" experiments for Windows, and also was not able to find a GATT client application for Windows that would allow the connection to last more than 30 seconds, so data was collected in a residential building environment, and each connection was lasting only 30 seconds, but it was still enough to collect some data. The Wi-Fi environment during the test is presented on Figure 53

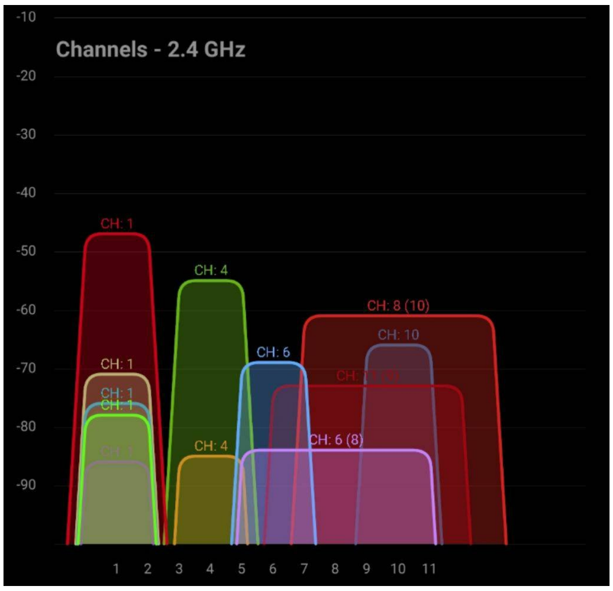


Figure 53. *Wi-Fi environment during the test captured with NetSpot*

Results:

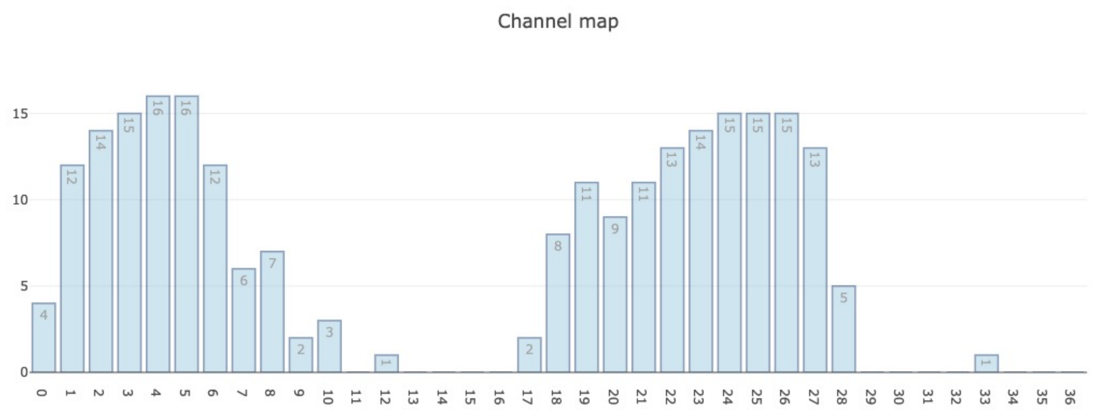


Figure 54. *Windows channel map by a router*

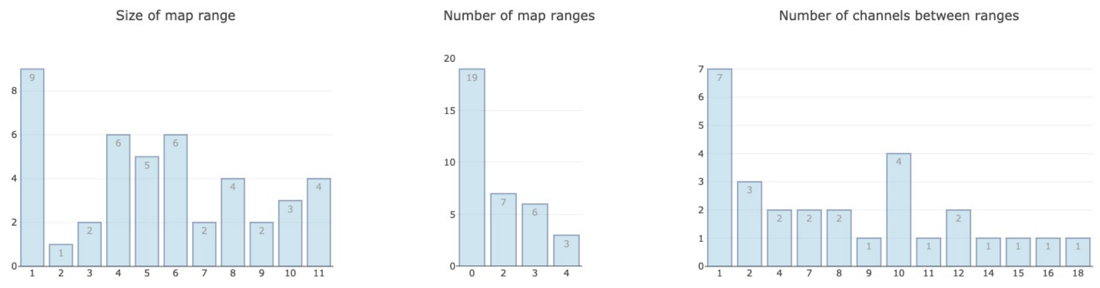


Figure 55. *Windows map ranges by a router*

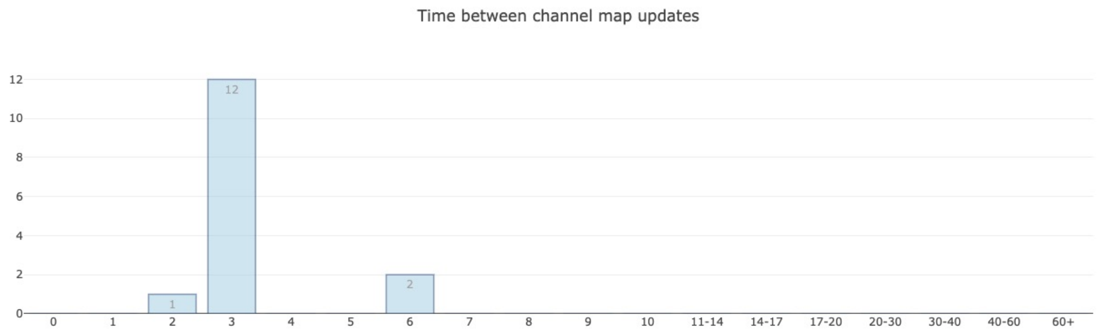


Figure 56. *Windows time between map updates by a router*

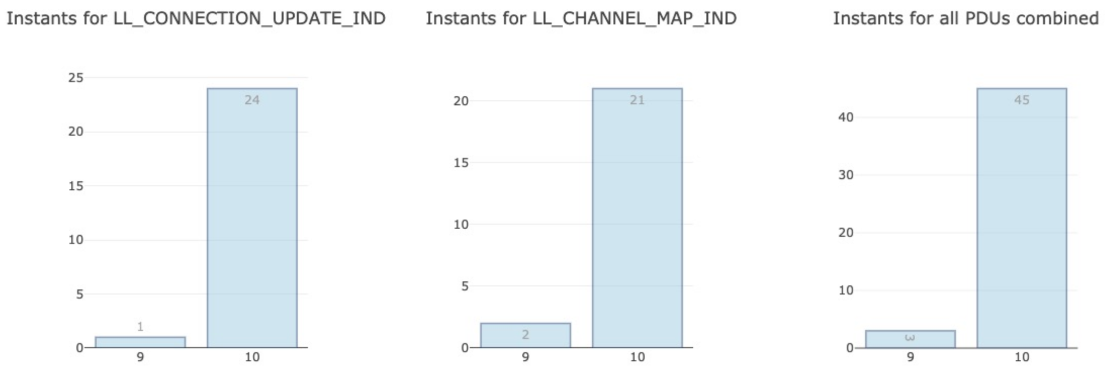


Figure 57. *Windows instants combined*

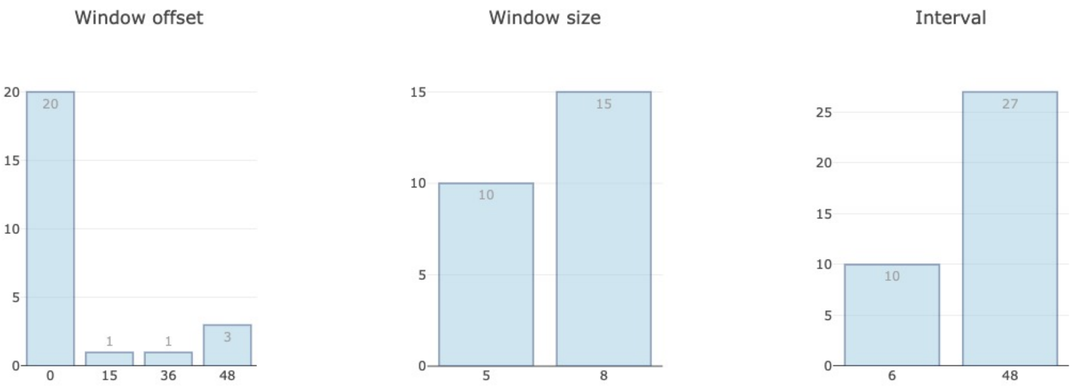


Figure 58. *Windows connection parameters combined*

The data was retrieved from a total of around 6 minutes of captured traffic that involved 12 connection initiations. Both the most frequent and the biggest instant was 10. It was used 94% of the time, and the rest was Interval 9.
Neither 2M, nor Coded PHYs have been observed to be used. Window offset varies in size, but for Window size, only 2 different values have been observed: 5 and 8.
Window offset varies in size, but mostly was equal to 0. 2 different values were observed for Window size: 5 and 8. Interval was never observed to be different from 48 or 6.
Author could not perform the test described in Chapter 3.5.3, so it is unknown if repetitively sending data can affect interval value on Windows.
Like for Android (Chapter 3.6.3) - connection was always initialized with a big Interval value. Then, within 0.5 - 3 seconds Interval changed to 6, the devices exchanged GATT characteristics in 5-10 seconds, and then Interval was updated back to the original value. The only difference from android is the value of the initial interval. It was 48, instead of 36.
One of the 12 connection initiations was noticed to have 2 ranges disabled on the channel map.

The high spike on Wi-Fi channel 4 (Figure 53) was supposed to lead to disabling 9-13 BLE channels, but has not (Figure 54)) and the reason for that is not clear. Data for Windows was collected using the same hardware located at the same spot as Linux test and their channel maps correlate with each other (Figure 47)).

3.11 Summary

One of the common characteristics that all platforms share is that none of them have used Coded PHY even once. This fact makes it safe to assume that whenever a control PDU with a 5-byte length specifier in the header is observed, it means that if the current PHY used is 1M - switch to 2M is about to happen and vice versa.

Windows, Mac, and Linux were found to be using 1M PHY only.

LL_CONNECTION_UPDATE_IND connection parameters were not found to be reacting to the amount of data exchanged between devices, and the only situation when Interval got changed - was GATT parameter exchange. The Interval value got decreased to 6 for a short period of time (5-10 seconds) and then restored to its original value. GATT parameter exchange was starting within 0.5 - 3 seconds from the initial connection on every platform. Interval value decrease happened on every platform besides iOS and Mac. iOS and Mac were using constant Interval values for every connection all the time.

Channel map on every platform was found to be highly influenced by surrounding Wi-Fi RF Noise.

4. Development and evaluation

4.1 LL_PHY_UPDATE_IND

Successful maintaining of synchronization has several prerequisites:

- fact of LL_PHY_UPDATE_IND sending has to be identified
- new PHY mode has to be guessed
- Instant should be guessed

We know from the collected data that no platform is using Coded PHY, therefore observing this packet with a 100% probability means switching from 1M to 2M PHY or vice versa. The current PHY mode for a connection is always known, so Sniffle will have to switch to the other one.

LL_PHY_UPDATE_IND is the only control PDU that has a 5-byte length, so whenever a 9 byte (5-byte payload + 4-byte MIC) long encrypted control PDU is observed, it represents LL_PHY_UPDATE_IND with 100% probability

We know from the collected data that Instant value across all platforms is within the 6-12 range. Values are not evenly distributed on the range, and each platform has an obvious most popular value. The most popular value is not the same for every platform, so we need to somehow identify a platform.

A wrong guess of the Instant value will not lead to losing synchronization with the connection, but would mean that abs(correctInstant - guessedInstant) number of hops are missed.
The collected data showed that we can identify a platform by Interval value:

- Interval 24 is only used by iOS
- Interval 12 is only used by Mac
- Interval 48 is only used by Windows
- Interval 36 is shared by Android, Gabeldorsche, and Linux

Instant value can be guessed according to the platform. Although Android, Gabeldorsche, and Linux can not be distinguished by this method - luckily, it is not required to pick the best Instant: Linux can be ignored since it only uses 1M PHY and both Android's and Gabeldorsche's most frequent and therefore most probable Instant is 9.

- For Android - Instant 9 would mean an 87.5% chance of not missing a single hop
- For Gabeldorsche - Instant 9 would mean a 94% chance of not missing a single hop
- For iOS - Instant 8 would mean a 95% chance of not missing a single hop

4.2 LL_CONNECTION_UPDATE_IND

For successful maintaining of synchronization:

- fact of LL_CONNECTION_UPDATE_IND sending has to be identified
- new Interval has to be measured
- Instant value should be guessed
- Window offset existence has to be addressed, since it shifts anchor point
- Window size existence has to be addressed, since may shift anchor point

Although, according to the BT core spec [25], LL_CONNECTION_UPDATE_IND is not the only control PDU that has 12-byte payload length - as mentioned in the chapter 1.5 - BT v5.2 is not widespread yet. It means that the fact of LL_POWER_CONTROL_RSP and LL_POWER_CHANGE_IND packet existence can be ignored for now. We assume that whenever a 16 byte (12-byte payload + 4-byte MIC) long encrypted control PDU is observed - it is LL_CONNECTION_UPDATE_IND and we should react accordingly.

If power control features in upcoming years become an obstacle in the identification of LL_CONNECTION_UPDATE_IND packets - Sniffle can look into FEATURE_RSP (Ex: Figure 11) packet content. It represents a list of features that devices support. If any of the 2 communicating devices have "0" flag for the "LE Power Class 1" feature - it means that current identification approach is still applicable for the connection.

Every packet has a "More Data" flag in its header, and it does not get encrypted, since headers are always exchanged in plain text. More Data flag indicates if there is any more data to be transferred on the current channel. When the value is False - it is safe to start listening to the next channel in the channel sequence.
Instant hopping behavior - when Sniffle switches to listening data from the next channel in the sequence as soon as the More Data flag is received and does not work

according to Interval value anymore - can be activated with a guessed Interval after LL_CONNECTION_UPDATE_IND packet is received.
Instant hopping will address issues created by Window offset and Window size, since it will basically be waiting on the channel and waiting for the data to be received. After that, it would require just several successfully captured packets to be able to calculate new Interval value from packet reception timestamps.

The downside of instant hopping is that in case some packet gets lost, Sniffle will wait forever for the packet to be received. It will result in synchronization loss. Generally, sniffing should happen according to the hopping Interval, but it is acceptable to switch to instant hopping mode for a short period of time.

An optimal Instant has to be guessed in order to minimize the time spent in instant hopping mode. The goal is to minimize the risk of losing synchronization due to a lost packet.

When guessing an Instant for LL_CONNECTION_UPDATE_IND, it is important to consider the risks of a wrong guess:
In case the guessed value is smaller than the actual one - Sniffle will spend more time in instant hopping mode than is necessary and in case a packet gets lost - lose synchronization.
In case the guessed value is bigger than the actual one Window offset will come before the instant hopping mode gets turned on and Sniffle will lose the synchronization right away.
Therefore it is safer to go with a smaller guess, since a bigger guess is a guaranteed failure.
Same principle as described in Chapter 4.1 can be used to determine the platform by current Interval.

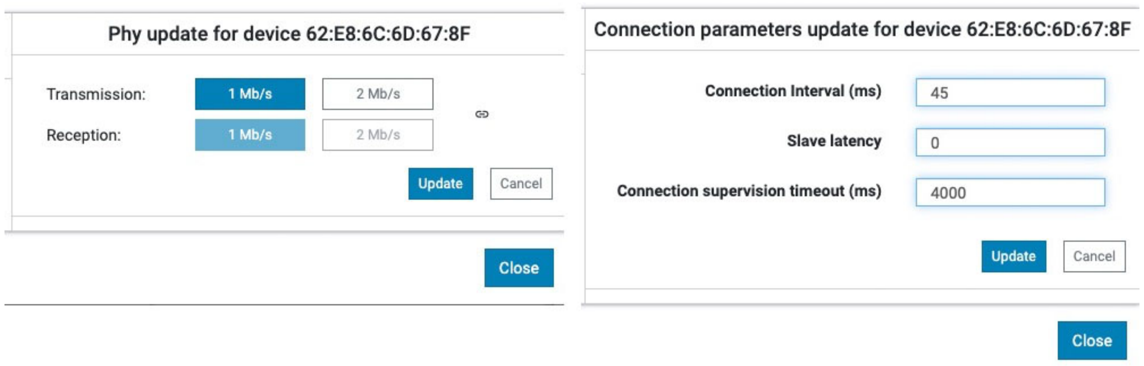
For Interval 36 - the best guess of Instant would be 6. Even though Instant 6 was observed only on Linux, that is a less popular platform than Android, and only 0.2% of times - immediate failure is too big of a risk, and it overweights the possibility of losing a packet.
If the environmental conditions lead to packet loss - there are plenty of other reasons that may lead to losing synchronization.
For Interval 24 - the best guess is 6
For Interval 12 - the best guess is 6
For Interval 48 - the best guess is 9
For Interval 6 - the best guess is 6.

4.3 Evaluation

Implementation of the solution on the Sniffle platform can be found on the following links:
https://github.com/nccgroup/Sniffle/commit/c3ee3b20363ad2d628fb7696e7e6114f530182ba

https://github.com/nccgroup/Sniffle/commit/4b9fa07b752ab3ae8a883f9ff4898a958bf1eefe

Since this paper is not addressing the channel map update issue - synchronization will be lost when LL_CHANNEL_MAP_IND packet gets transmitted. Therefore it is not possible to test this solution on the same platforms that were used for data collection. Nordic Semiconductor nRF52840 Dongle[1] was used in combination with nRF Connect Desktop[2] application in order to confirm that the implementation works. What allows nRF52840 to be used for evaluation is that it does not perform channel map updates at all. It constantly works with all channels activated. Another feature that makes testing convenient is that nRF Connect Desktop allows manual triggering of PHY and connection parameter updates. It also allows controlling certain values (Figure 59).


Phy update for device 62:E8:6C:6D:67:8F

Transmission: 1 Mb/s (selected) | 2 Mb/s

Reception: 1 Mb/s (selected) | 2 Mb/s

Connection parameters update for device 62:E8:6C:6D:67:8F

Connection Interval (ms): 45

Slave latency: 0

Connection supervision timeout (ms): 4000

Figure 59. *nRF Connect Desktop PHY and connection parameter update windows*

nRF52840 was used as a master, and OnePlus 8 Pro was used as a slave. Devices got paired, since pairing enables encryption.

After pairing LL_PHY_UPDATE_IND was triggered 50 times by switching from 1M to 2M PHY and back from 2M to 1M 25 times. Sniffle was able to identify every event and successfully react to them without losing synchronization.

After pairing LL_CONNECTION_UPDATE_IND was triggered 60 times in total. Interval 36 (36 * 1.25 = 45ms) was switched to 6 and back from 6 to 36 - 10 times. Then Intervals 12, 24 and 48 were set 10 times in rotation. Then random Intervals in the range between 15 and 50 were set 10 times. Sniffle was able to identify every event and successfully react to them without losing synchronization.

[1]nRF52840 Dongle `https://www.nordicsemi.com/Software-and-tools/Development-Kits/nRF52840-Dongle`

[2]nRF Connect Desktop `https://www.nordicsemi.com/Software-and-tools/Development-Tools/nRF-Connect-for-desktop`

5. Conclusion and future work

This study fulfills the objectives we set in the introduction part:

- Collected and analyzed data for popular Bluetooth stack implementations.
- Found characteristics that were useful for solving connection parameter and PHY mode updates.
- Provided reliable solution for connection parameter and PHY mode updates.
- Found characteristics that will be useful for future work - solving channel map updates.

Furthermore, the study successfully answered research questions asked in the introduction:

- Is it possible to affect connection parameters by controlling the data that is being sent between two devices?
- Is it possible to identify data exchange patterns, that were able to trigger connection parameter updates, by observing the connection?
- Is it possible to affect channel mapping by producing RF noise?
- Is it possible to correlate Wi-Fi RF noise with BLE channel map?
- Which PHY modes are commonly used and what is the general pattern of switching between modes?
- How effective is the solution for recovering LL_CONNECTION_UPDATE_IND parameters?
- How effective is the solution for recovering LL_PHY_UPDATE_IND parameters?

None of the tested Bluetooth stacks' connection parameters were found to be affected by the amount of transferred data. There was one scenario when connection interval got updated - right after the initial connection, it would change to 6 to exchange GATT characteristics and then soon back to its original value.

Channel map was found to be highly affected by surrounding Wi-Fi RF noise, and channel map was found to be correlating with it.

Coded PHY has never been observed to be used and whenever PHY mode change has occurred - it was either change from 1M to 2M PHY or vice versa.

Provided PHY mode update solution with roughly 90% probability does not miss a single packet and misses not more than 1-3 hops in less lucky cases.

Provided connection parameter update solution does not miss packets on the merit of instant hopping, and the risk of malfunction is minimized by smart instant selection.

For further research, it would be interesting to look into tapping into long-lived connections. Existing solutions do not take into consideration the possibility of channel map updates and hopping interval changes. The observed frequency of channel map updates is a critical obstacle for current solutions. They treat the collected data as reliable, while it can not be trusted to the fullest. This issue has to be somehow addressed.

Bibliography

[1] A. Nordrum. “Popular Internet of Things Forecast of 50 Billion Devices by 2020 Is Outdated”. In: (2020). URL: https://spectrum.ieee.org/tech-talk/telecom/internet/popular-internet-of-%20things-forecast-of-50-billion-devices-by-2020-is-outdated/.

[2] Bluetooth Sig. *2020 Bluetooth Market Update*. 2020. URL: https://www.bluetooth.com/wp-content/uploads/2020/03/2020_Market_Update-EN.pdf.

[3] G. Ho et al. “Smart locks: Lessons for securing commodity internet of things devices”. In: *Asia Conference on Computer and Communications Security*. 2016, pp. 461–472.

[4] A. Rose and B. Ramsey. *Picking bluetooth low energy locks from a quarter mile away*. DEF CON 24, 2016.

[5] A. Levi et al. “Relay attacks on bluetooth authentication and solutions”. In: *International Symposium on Computer and Information Sciences* (2004), pp. 278–288.

[6] J. Clulow et al. “So near and yet so far: Distance-bounding attacks in wireless networks”. In: *European Workshop on Security in Ad-hoc and Sensor Networks* (2006), pp. 83–97.

[7] M. Ryan. “Bluetooth: With Low Energy comes Low Security”. In: *USENIX WOOT* (2013), p. 4.

[8] Damien Cauquil. *BtleJack: a new Bluetooth Low Energy swiss-army knife*. URL: https://github.com/virtualabs/btlejack.

[9] M. Ossmann. *Ubertooth*. URL: https://github.com/greatscottgadgets/ubertooth.

[10] Damien Cauquil. *BtleJuice Framework*. URL: https://github.com/DigitalSecurity/btlejuice.

[11] Slawomir Jasek. *GATTacker*. URL: https://gattack.io/.

[12] Mike Ryan. *Crackle*. URL: https://github.com/mikeryan/crackle.

[13] Sultan Qasim Khan. *Sniffle*. URL: https://github.com/nccgroup/Sniffle.

[14] C. Wohlin. “Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering”. In: *18th International Conference on Evaluation and Assessment in Software Engineering*. 2014.

[15] Bluetooth. “Reporting Security Vulnerabilities”. [Accessed 13-12-2020]. 2020. URL: https://www.bluetooth.com/learn-about-bluetooth/key-attributes/bluetooth-security/reporting-security/.

[16] D. Spill and A. Bittau. “BlueSniff: Eve Meets Alice and Bluetooth”. In: *USENIX Workshop on Offensive Technologies (WOOT)* 7 (2007), pp. 1–10.

[17] W. Albazrqaoe, J. Huang, and G. Xing. “Practical bluetooth traffic sniffing: Systems and privacy implications”. In: Annual International Conference on Mobile Systems, Applications, and Services, 2016.

[18] D. Antonioli, N. O. Tippenhauer, and K. Rasmussen. “BIAS: Bluetooth Impersonation AttackS”. In: *IEEE Symposium on Security and Privacy*. 2020, pp. 549–562.

[19] D. Antonioli et al. “BLURtooth: Exploiting Cross-Transport Key Derivation in Bluetooth Classic and Bluetooth Low Energy”. 2020.

[20] D. Antonioli, N. O. Tippenhauer, and K. Rasmussen. “Key Negotiation Downgrade Attacks on Bluetooth and Bluetooth Low Energy”. 2019.

[21] M. Ryan. *Bluetooth Hacking: Tools And Techniques*. Ed. by hardwear.io. 2019. URL: https://youtu.be/8kXbu2Htteg.

[22] S. Sarkar, J. Liu, and E. Jovanov. *A Robust Algorithm for Sniffing BLE Long-Lived Connections in Real-time*. IEEE Global Communications Conference, 2019.

[23] M. Cominelli, P. Patras, and F. Gringoli. *One GPU to Snoop Them All: a Full-Band Bluetooth Low Energy Sniffer*. Mediterranean Communication and Computer Networking Conference, 2020.

[24] E. Park, M.-S. Lee, and S. Bahk. “AdaptaBLE: Data Rate and Transmission Power Adaptation for Bluetooth Low Energy”. 2019.

[25] Bluetooth Cig. *Bluetooth Core Specification Version 5.2 Feature Overview*. 2020. URL: https://www.bluetooth.com/wp-content/uploads/2020/01/Bluetooth_5.2_Feature_Overview.pdf.

[26] C. F. Chiasserini and R. R. Rao. “Coexistence mechanisms for interference mitigation between IEEE 802.11 WLANs and Bluetooth”. In: *21st Annu. Joint Conf. IEEE Comput. Commun. Societies*. 2002, pp. 590–598.

[27] A. Hsu et al. “Enhanced adaptive frequency hopping for wireless personal area networks in a coexistence environment”. In: *IEEE Global Telecommun. Conf.* 2007, pp. 668–672.

[28] T. M. Taher, K. Rele, and D. Roberson. “Development and quantitative analysis of an adaptive scheme for Bluetooth and Wi-Fi co-existence”. In: *6th IEEE Consum. Commun. Netw. Conf.* 2009, pp. 1–2.

[29] J. Li and X. Liu. “A frequency diversity technique for interference mitigation in coexisting Bluetooth and WLAN”. In: *IEEE Int. Conf. Commun.* 2007, pp. 5490–5495.

[30] Leonardo Töpsch. *Design and Implementation of Adaptive Frequency Hopping for Bluetooth Low Energy - Evaluation for High Throughput Applications*. 2020.

[31] D. Cauquil. “Defeating Bluetooth Low Energy 5 PRNG for Fun and Jamming”. In: *DEF CON 27* (2019). URL: `https://youtu.be/wkIdpK7mAk4`.

[32] Janggoon Lee, Chanhee Park, and Heejun Roh. *Revisiting Bluetooth Adaptive Frequency Hopping Prediction with a Ubertooth*. 2021.

[33] J. G. del Arroyo et al. “Enabling bluetooth low energy auditing through synchronized tracking of multiple connections”. In: *International Journal of Critical Infrastructure Protection, vol. 18*. 2017, pp. 58–70.

[34] Damien Cauquil. *BtleJuice: the Bluetooth Smart Man In The Middle Framework*. GreHack, 2016. URL: `https://youtu.be/KHkRsYnAWYo`.

[35] Sławomir Jasek. *GATTACKING BLUETOOTH SMART DEVICES*. URL: `http://gattack.io/whitepaper.pdf`.

[36] Sultan Qasim Khan. *Sniffle: A low-cost sniffer for Bluetooth 5*. hardwear.io, 2019. URL: `https://youtu.be/nClZzdvGlKg`.

[37] Nordic Semiconductor Asa. *nRF Connect for Mobile*. URL: `https://play.google.com/store/apps/details?id=no.nordicsemi.android.mcp`.

[38] Synapse Product Development. *BlueSee BLE Debugger*. URL: `https://apps.apple.com/us/app/bluesee-ble-debugger/id1336679524`.