Bluetooth Packet Capture and Analysis Using Wireless Product Testbed

by

Son Nguyen

A Thesis
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Master of Science
in
Electrical and Computer Engineering
by

December 2020

APPROVED:

Professor Alexander M. Wyglinski, Advisor

Professor Kaveh Pahlavan, Committee Member

Professor Emmanuel O. Agu , Committee Member

Abstract

Bluetooth has solidified its place as one of the most prominent and versatile wireless communication technology alongside Wi-Fi. In spite of its widespread usage, contemporary Bluetooth testing technology still has many unaddressed shortcomings and issues. Hardware limitations, computational complexity, and the lack of standardization are some of the longstanding problems. Furthermore, very few systems possess a proven and reliable capability to test both Wi-Fi and Bluetooth. This thesis seeks to develop an adaptive Bluetooth test framework that can not only overcome limitations with existing systems but can also accommodate other technologies and protocols as well. The work presented in this thesis included: a packet processing and logging utility, a promiscuous packet capture subsystem built around a production test engine, and a complementary subsystem that handle encryption and other security mechanisms.

Acknowledgements

I would like to express my deepest gratitude to my advisor Professor Alexander M. Wyglinski for his continuous guidance and support through both my undergraduate and master program. I am very thankful for the opportunity to work with him in the WILab at Worcester Polytechnic Institute.

I want to thank Professor Kaveh Pahlavan and Professor Emmanuel O. Agu for serving on my committee and providing valuable suggestions and comments with regards to my thesis.

I would like to thank the staff and engineers at octoScope giving me the opportunity to work on this project and for assisting me with various technical issues as well as for their financial support throughout the course of this project.

A special thank you goes to Galahad Wernsing, my colleague at the joint octoScope-WILab project, who developed many of the utilities and tools that the work presented in this thesis is built upon. I also want to thank my family, friends, and other colleagues at WILab for supporting and encouraging me throughout the course of this project.

Contents

List of Figures

List of Tables

List of Acronyms

ADC IND	Advertising Connectable Indirected
ADV DIRECT IND	Advertising Connectable Directed
ADV NONCONN IND	Advertising Non-Connectable Indirected
ADV SCAN IND	Advertising Scannable Indirected
AGC	Automatic Gain Control
API	Application Programming Interface
BLE	Bluetooth Low Energy
BR	Basic Rate
BR/EDR	Basic Rate/Enhanced Data Rate
CAC	Channel Access Code
CI	Coding Indicator
CONNECT REQ	Connect Request
CRC	Cyclic Redundant Check
DAC	Device Access Code
dBm	Decibel-Miliwatts
DPSK	Differential Phase Shift Keying

EDR	Enhanced Data Rate
EPB	Enhanced Packet Block
EVB	Evaluation Board
FDMA	Frequency Division Multiple Access
FHSS	Frequency Hopping Spread Spectrum
GAP	Generic Access Profile
GATT	Generic Attribute Profile
GFSK	Gaussian Freuquency-Shift Keying
GHz	Giga-Hertz
GMT	Greenwich Mean Time
GPIO	General-Purpose Input/Output
HCI	Host Controller Interface
IAC	Inquiry Access Code
IC	Intergrated Circuit
IDB	Interface Description Block
IEEE	Institute of Electrical and Electronics Engineers
IO	Input/Output
IoT	Internet-of-Things
ISM	Industrial, Scientific and Medical
kbps	Kilobits-per-second
kb	Kilobyte

L2CAP	Logical Link Control Adaptation Protocol
LE	Low Energy
LFSR	Linear Feedback Shift Register
MB	Megabyte
Mbps	Megabits-per-second
MHz	Mega-Hertz
MIC	Message Integrity Check
OPP	Object Push Profile
OS	Operating System
OSI	Open System Interconnection
PCAP	Packet Capture
PCAPNG	Packet Capture Next Generation
PDU	Protocol Data Unit
QPSK	Quadrature Phase Shift Keying
RFCOMM	Radio Frequency Communication
RS	Recommneded Standard
SCAN REQ	Scanning Request
SCAN RSP	Scanning Response
SCPI	Standard Commands for Programmable Instruments
SDR	Software Defined Radio
SHB	Section Header Block

SIG	Special Interest Group
TDMA	Time Division Multiple Acces
TERM1	Terminator 1
TERM2	Terminator 2
UART	Universal Asynchronous Receiver/Transmitter
UI	User Interface
USB	Universal Serial Bus
UTC	Coordinated Universal Time

Chapter 1

Introduction

1.1 Motivation

Among the technological advances of the Information Age, modern wireless telecommunication protocol and technologies such as Wi-Fi and Bluetooth have always been considered to be among some of the most important if the most important innovations of the era [11] [12] [13]. With the prevalent usage of smart devices such as smartphones, tablets, and IoT appliances, the importance of wireless technology in every facet of modern life has only increase [14] [15] [6]. Even though Wi-Fi has always traditionally been seen as the predominant wireless technology, Bluetooth has been experiencing exponential growth in usage in recent years [3] [2] [1]. Particularly, Bluetooth enabled device shipment has nearly double in the past 6 years, growing from 2.4 billion device shipped in 2013 to 4.6 billion devices shipped in 2019. Furthermore, this upward trend is expected to continue and future projection are optimistically estimating that annual shipment will eventually reach 6.2 billion Bluetooth enabled devices shipped in 2024 [3]. (see Figure 1.1)

Throughout its life cycle, Bluetooth and related derivative technologies have evolved beyond their original purpose. Aside from traditional application such as audio streaming and data transfer, Bluetooth-related technologies have also been utilized in healthcare, industrial, and IoT application among others [16]. From the Airpods earphones [17] to specialized applications used for COVID-19 contact tracing [18], Bluetooth technologies

truly have a ubiquitous presence in every facet of the economy.

Even though Bluetooth has grown to rival Wi-Fi in both usage and versatility [19] [20], there remains a gap in the testing capability between the two type of wireless communication technology. In particular, whereas there is a wide variety of reliable testing hardware and software for Wi-Fi, the options for proven Bluetooth testing solution are still quite limited [21]. As many smart devices utilize both Wi-Fi and Bluetooth, there is considerable demand in the market for a reliable testbed that can test both type of wireless technology simultaneously [22]. Despite the significant level of interest, dependable solutions suitable for testing both Wi-Fi and Bluetooth are few and far between [23]. The work detailed in this thesis pertain specifically to the development of the Bluetooth component of a project started at octoScope to address the aforementioned gap in the market [24].

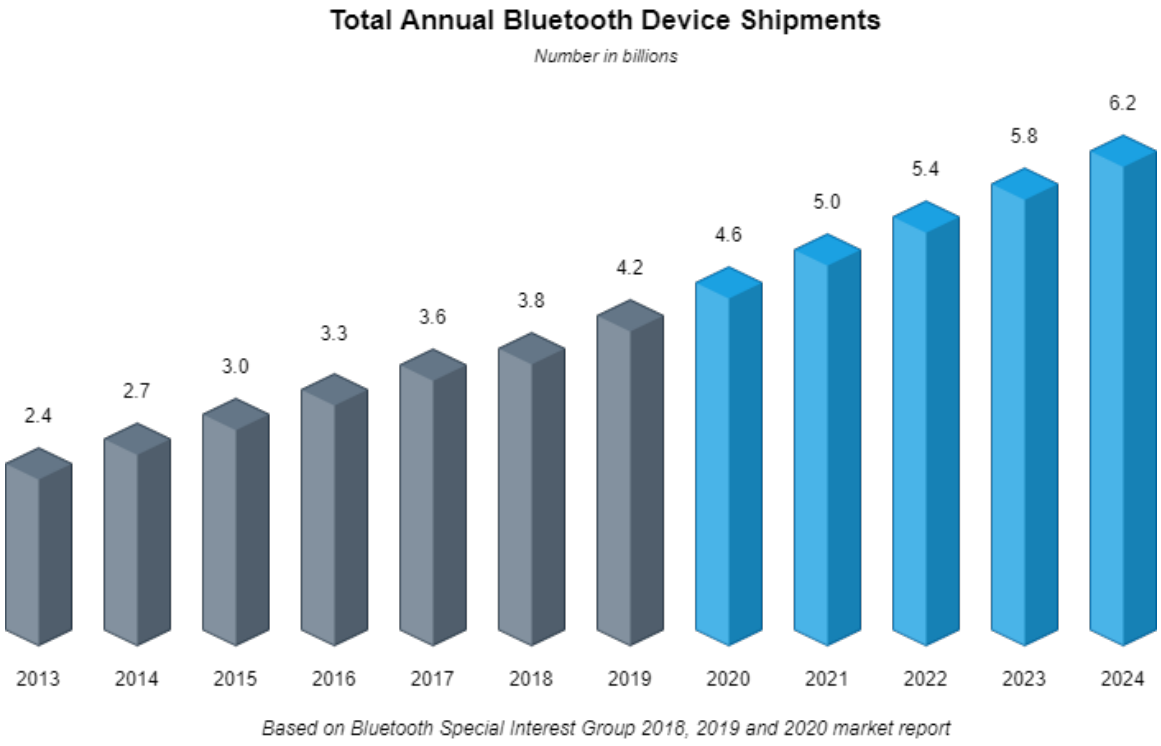


Figure 1.1: Bar graph showcasing the trajectory of growth in annual Bluetooth device shipment, with grey portion detailing annual Bluetooth device shipment growth from 2013 to 2019 and the blue portion describing projected growth from 2020 onward to 2024. The data used to compile this graph was collected from Bluetooth SIG's market report in 2018, 2019, and 2020 [1] [2] [3]

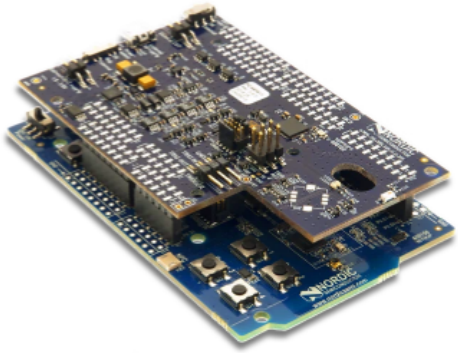
1.2 State of the Art

Generally speaking, the underlying working principle behind all wireless telecommunication test systems are fundamentally the same. Irrespective of the technology or protocol, an effective wireless test system must be able to isolate, monitor, decode, and log traffic from the test device. Bluetooth test systems generally follow the same formula [25]. Although not as numerous as those dedicated to Wi-Fi testing, a widespread availability of Bluetooth test systems does exist. Many of these systems and tools were first designed and developed with Wi-Fi testing in mind and limited Bluetooth testing capabilities were only added in later versions. However, with that said, it should be noted there are also some dedicated Bluetooth test system that are proven and widely used [23] [26]. Throughout the course of this thesis, the capabilities and limitations of these systems were studied and taken into consideration to help determine the design goal of our own system. From what was learned, Bluetooth test systems can generally be classified into three tiers based on the sophistication of their design.

The first tier consists of systems that are built around a single intergrated circuit (IC) and usually come in the form factor of a Universal Serial Bus (USB) dongle or a dedicated evaluation board (EVB). Some noteworthy first tier Bluetooth systems include the open source Ubertooth One/Zero [27] as well as some of the older offerings from Texas Instrument [28], Nordics Semiconductor [29], and Cypress Semiconductor [30]. Although systems of this tier are simplest and therefore the most affordable, they can usually only monitor a limited number of channels at a time and they cannot be easily updated to accommodate newer versions of Bluetooth. Systems in the second tier, which are more complex and more expensive, are actual quite similar to the first generation system in form factor and size. Second tier systems can monitor a large amount of channel at once but the ability to accommodate newer versions of Bluetooth are generally still limited due to hardware constraint. One example of a second tier system would be the Teledyne Lecroy Frontline ComProbe BPA [31] and more recent offerings from Nordics would also fit the definition [10]. Third-tier systems, which are built-around Software Defined Radio (SDR) [32], are the most capable and complex but also the expensive type of Bluetooth test systems. Furthermore,

third-tier systems not only possess the ability to monitor all the channels at the same time but also the ability to update the firmware to accommodate and support any future version of Bluetooth though the use of SDR [26] [33]. Some examples of third-tier systems include the Sodera series of Bluetooth Analyzer produced by Teledyne-Lecroy [34] as well as Ellisys's various offerings [35] [36]. Figure 1.2 showcases examples of sniffers from all three tiers.

(a) Ubertooth One [27]



(b) nRF52 DK Sniffer [29]

(c) Teledyne Lecroy SODERA Sniffer [34]

Figure 1.2: Examples of Tier 1, 2, and 3 Bluetooth sniffers. It is worth noting that there is a evident and noticeable difference in form factor's size between each tier of device owing to the increase in the complexity and number of components.

Generally, there is a lack of standardization when it comes to the file format used to log intercepted traffic. In most cases, captured Bluetooth traffic are logged into non-standard proprietary packet capture file formats designed by the manufacturer [26] [37]. Smart devices such as smartphones and tablets used BTSnoop files as a way keep record of Bluetooth data exchanges [38] [39]. While BTSnoop is a reliable format that is compatible with many packet capture software, it is only designed to contain certain type of Bluetooth packets. Furthermore, since BTSnoop was designed specially to accommodate Bluetooth transmission, it is incapable of handling other type of wireless traffic [39]. For systems that are designed to handle both Bluetooth and Wi-Fi, more universal and adaptive platform such as PCAP [40]and PCAPNG [41] are usually used instead.

While Bluetooth test systems of all tiers can more or less handle intercepting, analyzing, and logging packets on their own, even the most capable system has limited capacity to deal with interference. Instead, an isolated environment, usually in the form of a Faraday cage or an anechoic chamber (refer to Figure 1.3a and b), is used to prevent other signals from interfering with the transmission coming the test device. [42] Traditional Faraday cages and anechoic chamber are usually the size of a small room if not bigger, but semi-anechoic chambers such as those produced by octoScope (see figure 1.3c) comes in much smaller form factor ranging from the size of a small locker to that of a cupboard [43] [44]. Augmented by add-on hardware, these chambers can also be used to simulate real-world environments, allowing a wide variety of scenarios to be tested from the confine of one test chamber [43].

From an academic point of view, there has certainly been interest in Bluetooth packet capture. In [45], BLE packets was formatted and processed for wireless penetration testing and simulating a man in the middle attack. Moreover, in the book Bluetooth security [46], Yang and Huang specifically examined the PCAP format usage in a number of cybersecurity application. Mazzazenga, et al's work [47] does touch on packet capture with regards to testing, although the paper emphasis and focus on simulation and mathematical model instead of practical tests.

Regarding Bluetooth promiscuous sniffing, there has been a decent amount of academic work and research on topic. For instance, in [6], a Bluetooth sniffer was utilized for coexistence testing for industrial application in a factory floor. Aside from purely application

Table 1.1: Comparison between the existing academic literature, work and research and the work presented in this thesis on Bluetooth packet capture file formatting

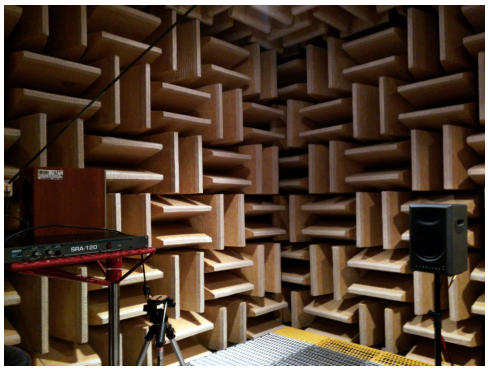
Existing Literature	Similarities	Differences
Taj Dini, et al. [45]	Focus on BLE Consideration about other 2.4GHz ISM technology Mentioned BLE packet recovery	Primarily focus on Bluetooth/ZigBee security rather than performance testing No mention of a specific capture file format
Yang, Huang [46]	Focus on Bluetooth (both BR/EDR & LE) Usage of Wireshark Usage of PCAP format	Usage of open-source hardware No mention of PCAPNG More emphasis on security rather than testing
Mazzazenga, et.al [47]	Focus on BR/EDR Focus on packet	Usage of simulation Emphasis on theoretical math model

related work, in [45], the authors built a SDR-based Bluetooth promiscuous sniffer for cybersecurity application using USRP in conjunction with gnuradio. While many articles examined the SDR-based approach like in [45], there are also articles such as [48], where a system based on inexpensive conventional Bluetooth-compliant radios are used to build a promiscuous Bluetooth sniffer instead.

Overall, the ideal Bluetooth test system would be one that possess not only adaptive yet reliable packet capture capability but also the ability to manipulate the test environment and log captured packets into universally compatible file format. Although only a few contemporary systems possess this combination, such systems represent the not only the pinnacle but also the future standard of Bluetooth testing [49].

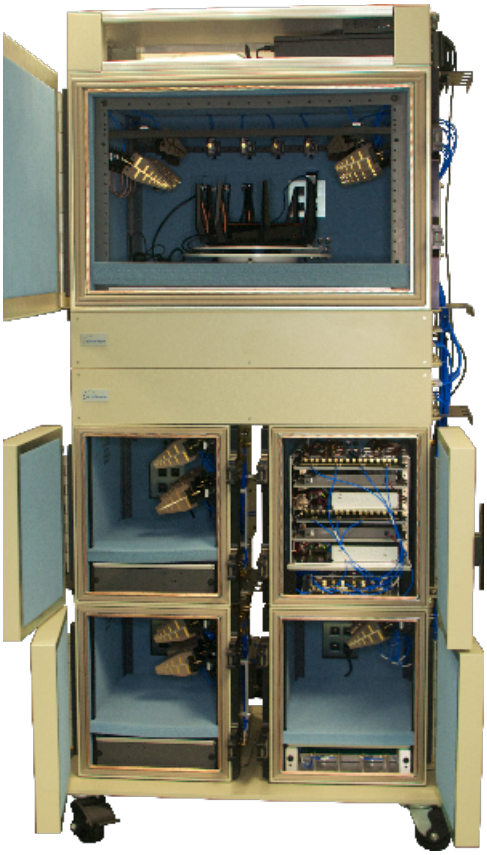
Table 1.2: Comparison between the existing academic work, literature, as well as research and the work presented in this thesis on Bluetooth promiscuous sniffing for testing

	Similarities	Differences
Wetzker, et al. [6]	Consider Bluetooth (both BR/EDR & LE) alongside Wi-Fi Usage of sniffer for testing purpose	Emphasis on Industrial application Factory floor used in test Emphasis on coexistence
Albazrqaoe, et al. [48]	Focus on Bluetooth (both BR/EDR & LE) Promiscuous/Passive Sniffing	Usage of inexpensive Bluetooth-compliant radio Emphasis on novel hardware design/architecture
Taj Dini, et al. [45]	Focus on BLE SDR-based sniffer Utilization of BLE Beacon in test	Usage of USRP w/ gnuradio Emphasis on Wireless penetration testing Man-in-the-middle attack



(a) Anechoic Chamber [50]

(b) Faraday Cage [51]



(c) octoScope Semi-Anechoic Box [44]

Figure 1.3: Examples of various types of equipment used for creating an isolated RF environment in wireless testing.

1.3 Current Issues

Despite recent advances, contemporary Bluetooth test systems still have a number of limitations and issues. While many of these problems affect all classes of Bluetooth test systems equally, some are only limited to certain classes and types of test system.

Hardware limitations are a particularly consistent issue that inhibit the performance and utility for lower-end and older systems. Most lower-end systems only possess the ability to monitor one channel at a time as monitoring multiple Bluetooth channels at the same time is a computationally complex and intensive task. While tests that involve a small number of devices will only involve a few channels, tests designed to emulate real-life scenarios will more often than not involve a large portion of all available channels. Therefore, the ability to run and conduct more complex tests with a large number of test devices are only reserved for the most sophisticated and advanced systems. Furthermore, hardware limitations on lower-end and older systems hamper these system's ability to be updated to support and accommodate newer versions of Bluetooth. Thus, the lifespan for lower-end systems are effectively shorten as they would be rendered obsolete every time a new version of Bluetooth is released [26].

Dealing with built-in Bluetooth security mechanism and encryption is not only a challenge for lower-end systems but for higher-end system as well. Most Bluetooth packets are encrypted and the useful data they contain is essentially locked away. In order to decrypt the packets, specific credentials need to be obtained. While the method used to brute force the credentials differ system to system, they are usually time consuming and prone to error, making them unsuitable for time sensitive and critical operations. [52]

Another issue that affects all Bluetooth test systems is the lack of a standard and universal packet capture file format. More often than not, captured Bluetooth traffic is logged into non-standard proprietary packet capture file formats that require specialized software to read. It is often complicated and difficult to export the captured data to another format. Furthermore, since these file formats are specifically designed to hold Bluetooth packets, they would be incompatible with other wireless protocol and technology. Thus, for systems that must handle both Wi-Fi and Bluetooth, these types of files cannot be

used [37] [53].

1.4 Thesis Contributions

The work detailed in this thesis is part of a larger project that seeks to build a test system that can not only provide both accurate capture of packets but also information regarding the RF environment through both inline and promiscuous packet capture, respectively, and then combine all the gathered data into a single packet capture file. Figure 1.4 showcase a prototype of the wireless test system developed at octoScope thatthe Bluetooth test utilities is built upon. Specifically, this thesis described the development process of the Bluetooth component of the aforementioned test system. Since many of the tools and utilities for the Bluetooth test system was developed in conjunction with a colleague at the joint octoScope-WILab project, the work presented in this thesis is complemented by the research presented in [54]. This work contains the following contributions:

1. A supplementary tool for the inline packet capture subsystem that can extract, process and logs both regular Bluetooth packets and Bluetooth HCI packets into standard file formats such as BTSnoop, PCAP and PCAPNG;

2. Technical detail and relevant test results and data along with solution to some of the issue encountered;

3. The promiscuous packet capture subsystem built around the Litepoint product test engine which is supposed to complement the inline packet capture by providing data about the RF environment as well as filling in what its counterpart may have missed;

4. Two versions of the subsystem built with the Litepoint IQxel-M16W and the Litepoint IQxel-MW 7G respectively;

5. Over-the-air capture test results and data for the two versions of the Litepoint based promiscuous packet capture subsystem and solutions to a number of issues;

6. Ongoing effort to develop a complementary promiscuous packet capture subsystem that will help the Litepoint-based system handle encryption and other built-in security

mechanism without relying on brute forcing;

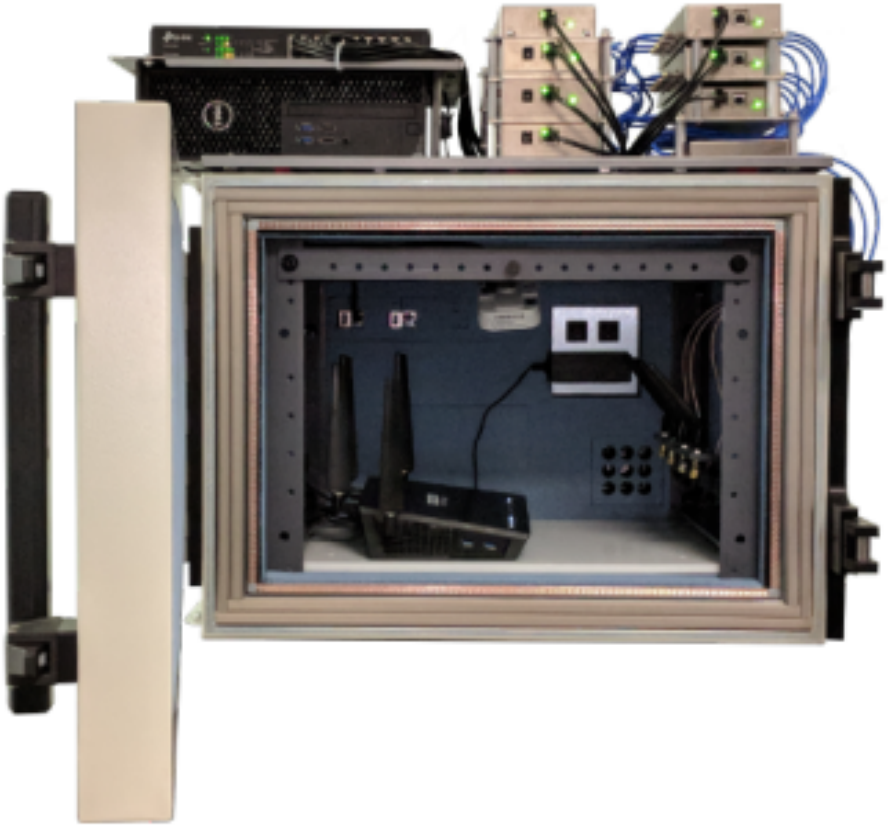


Figure 1.4: Early mockup prototype of the wireless test system developed at octoScope that the Bluetooth test utilities is built upon and is supposed to complement. [4]

1.5 Thesis Organization

The thesis is organized as follows: Chapter 2 presents background information on all variants of the Bluetooth protocol, contemporary Bluetooth testing systems and various packet capture file formats utilized in the project. Chapter 3 presents relevant technical detail and describe the development process for the packet formatting and logging tool. Additionally, the inline packet capture system that the tool is supposed to supplement is

also briefly described in this chapter. Chapter 4 focus on the development process of the promiscuous packet capture subsystem built around the Litepoint production test engine and the complementary decryption subsystem. In addition to technical data, both Chapter 3 and 4 also present relevant test results and data as well as solutions to the issues encountered during the respective development process. Finally, Chapter 5 discusses the outcome of the work done and outline the future direction of the project.

Chapter 2

Overview of Bluetooth and Packet Capturing Technology

Bluetooth and its various derivative technologies are members of the wireless technology standard group previously maintained by the Institute of Electrical and Electronics Engineers (IEEE) as IEEE 802.15.1 but later on standardized by the Bluetooth Special Interest Group (SIG) [16].

Figure 2.1: Image depicting a Recommended Standard (RS) 232 serial cable [5]. Originally introduced in 1960, RS-232 is primarily utilized for serial data transmission, and communication.

Table 2.1: Frequency allocation for 12 of ISM radio bands. The 2.4 GHz band utilized by Bluetooth and derivative technologies is highlighted in Cyan

Frequency Range	Center Frequency	Bandwidth
6.765 MHz - 6.795 MHz	6.78 MHz	30 kHz
13.553 MHz - 13.567 MHz	13.56 MHz	14 kHz
26.957 MHz - 27.283 MHz	27.12 MHz	326 kHz
40.66 MHz - 40.7 MHz	40.68 MHz	40 kHz
433.05 MHz - 434.79 MHz	433.92 MHz	1.74 MHz
902 MHz - 928 MHz	915 MHz	26 MHz
2.4 GHz - 2.5 GHz	2.45 GHz	100 MHz
5.725 GHz - 5.875 GHz	5.8 GHz	150 MHz
24 GHz - 24.25 GHz	24.125 GHz	250 MHz
61 GHz - 61.5 GHz	61.25 GHz	500 MHz
122 GHz - 123 GHz	122.5 GHz	1 GHz
244 GHz - 246 GHz	245 GHz	2 GHz

All versions of Bluetooth operate in the 2.400 to 2.485 GHz spectrum, also known as the Industrial, Scientific, and Medical (ISM) band, one of the most widely used frequency bands for wireless technology development [55] (see Table 2.1). While originally only intended to be utilized as a wireless substitute for the RS-232 data cables (see Figure 2.1), Bluetooth and its derivatives have evolved for use in not only communication but also healthcare and industrial applications [16] [6]. Overall, Bluetooth can be classified into two main technology standards: (1) Bluetooth Basic Rate/Enhanced Data Rate (BR/EDR) [16], and (2) Bluetooth Low Energy (LE) [16]. Bluetooth BR/EDR, also known as Bluetooth Classic, refers to the original implementation and its evolution while Bluetooth LE, also widely referred to as BLE, refers to the separate protocol adopted June 2010 [16]. Both the BR/EDR variant and the LE variant share many similarities regarding technological specifications but the latter is designed for use in Internet-of-Things (IoT) and sensor networks, where battery conservation is an important factor in the design process.

Packet capturing systems, also commonly known as sniffers [56], capture and record

traffic in a digital communication network. As data packets transit across the network, it is the sniffer's goal to capture, decode, analyze, and display the information the packets carry in human-readable formats [57]. A packet capturing system consists of two main component, as illustrated in figure 2.2: (1) the communication hardware that scan the airwaves for packets, and (2) the software that control the aforementioned hardware as well as handle the decryption and analysis of intercepted packets [56]. Generally, packet capturing systems refer to both systems built specifically for wired and wireless communications. Since this thesis primarily deals with Bluetooth, the main focus is on packet capturing systems for wireless network. As most wired network hardware have built-in functions that allow the user to monitor packets that pass through the network, it is a relatively straightforward task to capture network traffic over a wired network. Intercepting wireless communications represents a much more complex problem, especially for wireless technology such as Bluetooth with built-in frequency hopping features for security purposes [58].

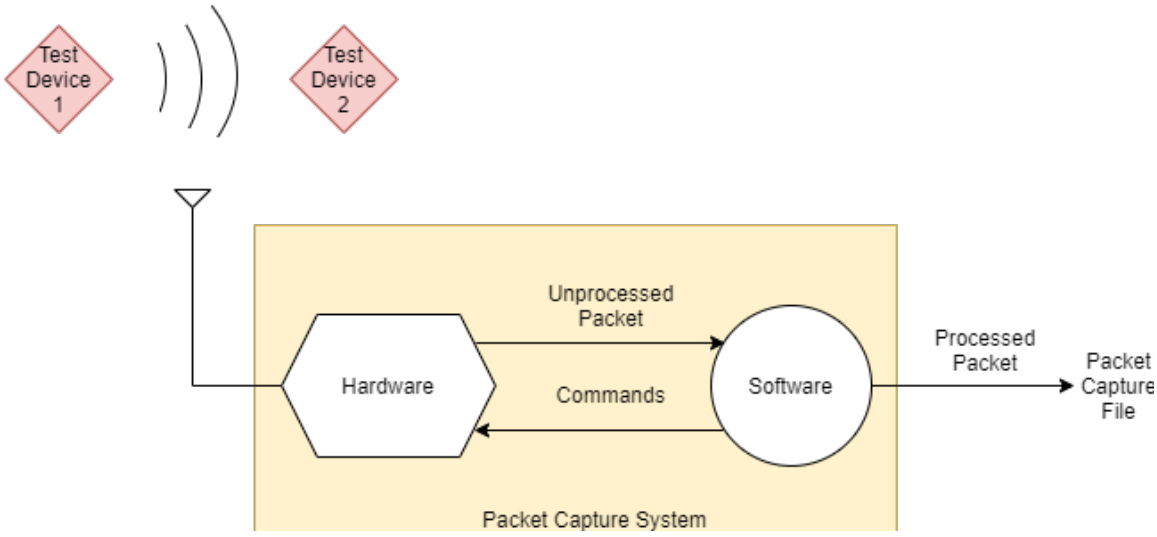


Figure 2.2: General packet capture system structure illustrating the software/hardware divide within the a sniffer system. While the hardware component scan the airwaves for packets, the software component handle decryption, decode , and control the hardware.

In this chapter, we provide an overview on the Bluetooth wireless technology standard, with a focus on Bluetooth LE, as well as packet capturing systems. Technical details and specifications on the Bluetooth wireless technology are discussed in Section 2.1 to provide

context and knowledge needed in discussion of Chapter 3. Packet capturing systems, which constitute both hardware and software, is discussed in Section 2.2. Furthermore, relevant details about data formats used in packet capturing in Section 2.3 is to support discussion in both Chapter 3 and 4.

2.1 Bluetooth Wireless Technology Standard

As laid out in the Bluetooth core specifications, Bluetooth wireless communication standard is primarily geared towards short-range communication acting as substitute for cables intended to connect portable and/or fixed electronic devices. Bluetooth operates alongside other common wireless protocol like Wi-Fi (IEEE 802.11) and ZigBee (IEEE 802.15.4) in the 2.4 GHz band, specifically between 2.402 and 2.480 GHz or 2,400 and 2.4835 GHz, with 3.5 MHz and 2 MHz wide guard band at the top and bottom respectively [59] [60]. (see figure 2.3) Overall, Bluetooth systems can be classified into two classes: Bluetooth Basic Rate/Enhanced Data Rate (BR/EDR) devices and Bluetooth Low Energy (LE) [16]. While there are certainly many differences between BR/EDR and LE, there are also many similarities, as can be seen by the comparison in table 2.2. Built-in device discovery, connection establishment, and connection mechanisms are available for both classes of devices. Designed to be less expensive, less complex, and less power consuming than Bluetooth Basic Rate/Enhanced Data Rates system, BLE systems generally have lower data rates and duty cycles than Bluetooth Basic Rate/Enhanced Data Rate systems. Devices and systems that support both BLE and Bluetooth Basic Rate/Enhanced Data Rate should be able to communicate with any other device that supports one or both of the technologies, enabling the use of most profiles. However,it is worth mentioning that there are specific profiles that are only available with either BLE or Bluetooth Basic Rate/Enhanced Data Rate [16].

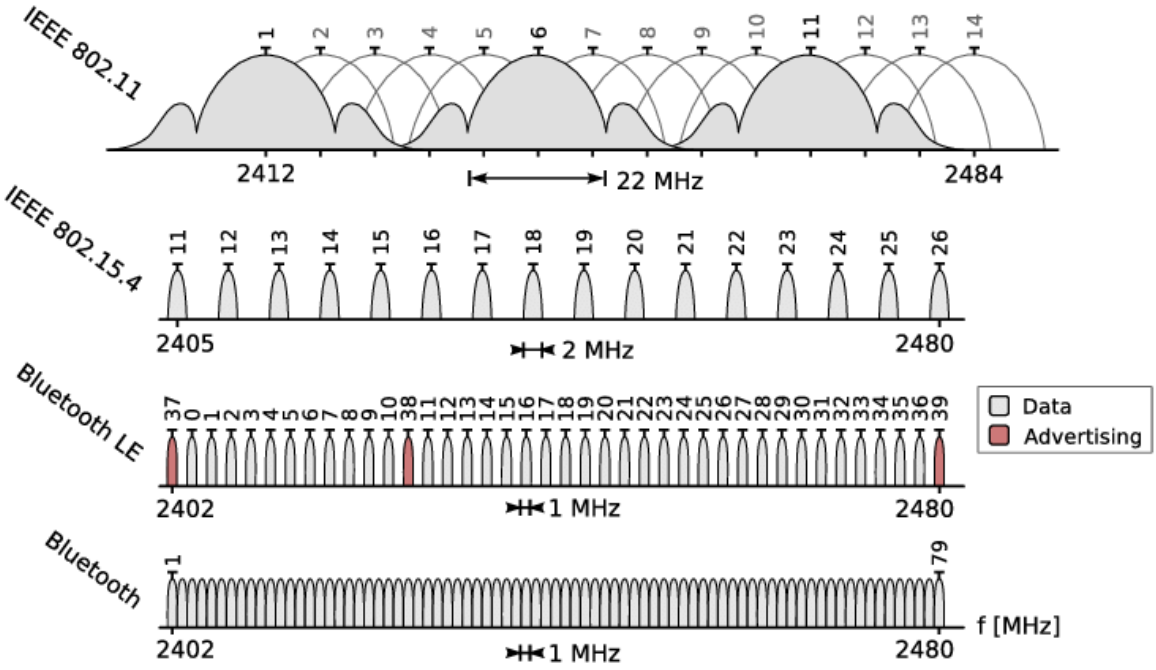


Figure 2.3: Channel maps of popular wireless telecommunication technologies and protocols utilizing the 2.4 GHz ISM band. [6]

Table 2.2: Comparison between the physical parameter of the BR/EDR and the LE variant of Bluetooth. Coding scheme, data rate, number of channels and hopping intervals are all considered to be major differences between BR/EDR and LE

	BR/EDR	LE
Frequency Range	2.4 GHz ISM	
Spread Spectrum	FHSS	
Coding Scheme	GFSK (BR) 4/8-DPSK(EDR)	GFSK
Data Rate	1 Mbps (BR) 2 or 3 Mbps (EDR)	1 Mbps
Number of Channels	79 (1 Mhz guard band)	40 (2 Mhz guard band)
Hopping Interval	625 μs	75 ms - 4s

In a typical Bluetooth core system, there is only one host but there might be more than one controller. The host in a Bluetooth core system encompasses all of the layers below the non-core profiles and above the Host Controller Interface (HCI), whereas a controller encompasses all the layers below the HCI [16]. Controllers are further classified into primary controllers and secondary controllers. A Bluetooth core systems many only have one primary controller, which may be a BR/EDR controller, an LE controller, or a combined BR/EDR and LE controller, but may have more than one secondary controller that are alternate MAC/PHY (AMP) controller.

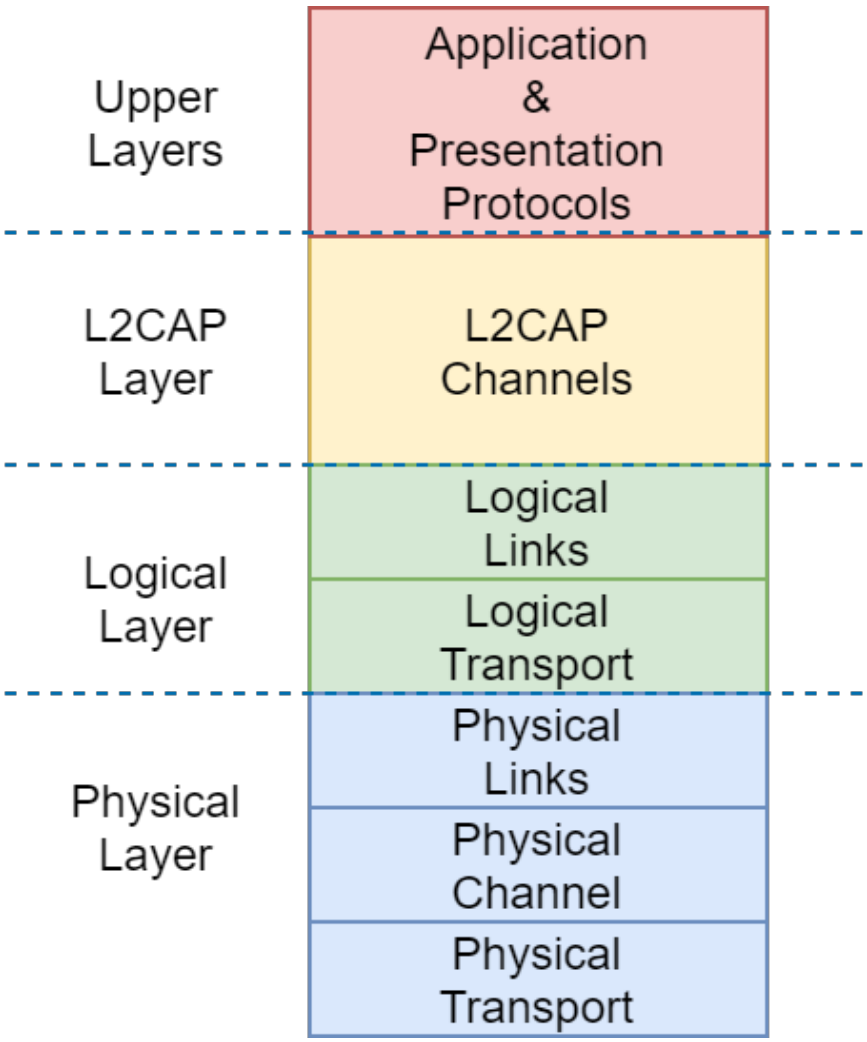


Figure 2.4: General stack structure for all Bluetooth derived technology, BR/EDR and BLE included

Overall, all variants of Bluetooth, including BR/EDR and LE, follow the same generic layered architecture which are subdivided into the Physical Layer, Logical Layer, and several Upper Layers [16] [61] (see Figure 2.4). While the Physical Layer and Logical Layer resides on the controller, the Upper Layers reside on the host. The Physical Layer always consists of physical transports, physical channels, and the physical links whereas the Logical Layer always consists of logical transports and logical links. The Upper Layers may differ depending on the application and variants of Bluetooth in question but the lowest layer is always the Logical Link Control Adaptation Protocol (L2CAP) channel.

2.1.1 Bluetooth BR/EDR

BR radios can theoretically support a nominal data rate of up to 1 Mbps with Gaussian Frequency-Shift Keying (GFSK) coding while EDR radios can accommodate a nominal data rate of up to 2 or 3 Mbps for Π/4-Quadrature Phase Shift Keying (π/4-QPSK) and 8-Differential Phase Shift Keying (8-DPSK) [59]. Bluetooth BR/EDR allows for full-duplex communication in the unlicensed 2.4 GHz ISM band. Since other communication technologies, most notably the widely adopted Wi-Fi, also occupies the 2.4 GHz ISM band, Frequency Hopping Spread Spectrum (FHSS) is utilized in Bluetooth Basic Rate/Enhanced Data Rate to prevent fading as well as destructive collision and interference [16] [6]. Generally, with FHSS, BR/EDR radios jump between the 79 allocated frequency (see Figure 2.5), each 1 MHz apart, in the 2.4 GHz ISM band in a pseudo-random pattern. However, in certain cases, sections of the frequency band that are being occupied by other devices can be specifically bypassed to avoid interference. A BR/EDR device may frequency hop between the transmission and reception of packet. Using both Frequency Division Multiple Access (FDMA) and Time Division Multiple Access (TDMA), a BR/EDR channel is partitioned in both the time and frequency domains into time slots so that one frequency can be occupied by different BR/EDR devices at different times [16]. In special cases, one packet may occupy multiple consecutive slots on the same frequency although in most cases one packet occupies several different slots over a number of distinctive frequencies.

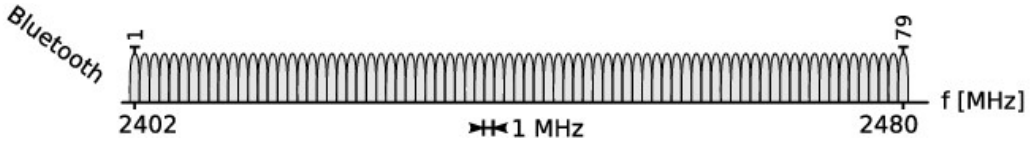


Figure 2.5: Bluetooth BR/EDR Channel Map. BR/EDR devices make use of 79 channels, each 1 MHz apart, to exchange data. [6]

A group of devices on the same BR/EDR physical channel generally synchronize and adhere to a similar clock rate and frequency hopping pattern in most cases. Devices in a typical BR/EDR operation can be classified as either master or slave. The master refers to the device that provides the common clock and frequency hopping pattern, whereas slaves refer to devices that adhere to these shared parameters [16]. A group consisting of one master and its slaves operating in this manner is called a piconet. Devices that are in the same piconet use the same specific frequency hopping pattern that is calculated based on the synchronized clock rate the master provides as well as certain fields within the BR/EDR packet.

In a physical channel, the link between a master device and its slave devices is generally referred to as a physical link. The physical link's purpose is to provide bidirectional packet transport between the master device and associated slave devices or to accommodate unidirectional packet transport for the master device to associated slave devices [16]. Due to the large amount of devices that may be within the confine of one physical channel, the ability to form a physical link is restricted and controlled. Specifically, only links between a master device and slave devices are permitted, whereas links between slaves devices are not allowed.

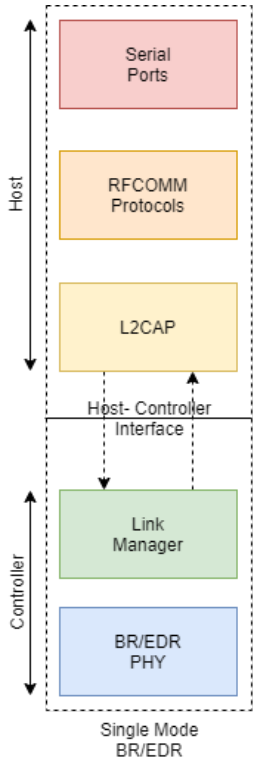


Figure 2.6: Example of a Typical BR/EDR Stack. The stack in this example follows the generic layered architecture with the BR/EDR upper layers consisting of serial ports and Radio Frequency Communication (RFCOMM) protocol.

Overall, as shown in Figure 2.6, the BR/EDR stack follows the generic layered architecture and can be partitioned into two main parts, the layers that inhabit the host and those that inhabit the controller with the HCI acting as intermediary between the host and the controller [61] [16]. The Logical Layer for BR/EDR is specified as the Baseband Layer. The Baseband Layer and its lower counterpart, the Physical Layer reside on the controller side while the Upper Layers reside on the host side. While the specific term for each layer of the Upper Layers are different, they generally fulfill the same function as that of the network, transport, session, presentation, and application layers in the Open System Interconnection (OSI) model.

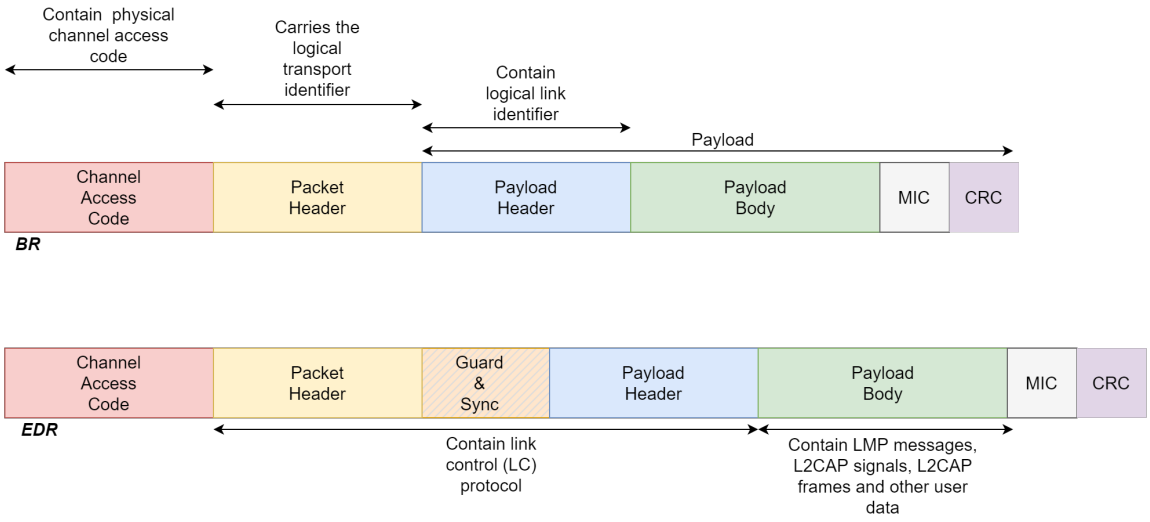


Figure 2.7: A typical BR packet alongside a typical EDR packet. Aside from the difference in data rate, the Guard & Sync field is another major difference between BR and EDR. The Guard & Sync field contains the guard period and the synchronization sequence.

The Generic BR/EDR packet consists of 3 main parts (shown in Figure 2.7): the access code, the header, and the payload [59]. The access code, which can either be 68 or 72 bit long, is used for timing synchronization, offset compensation, paging, and inquiry. There are three different kinds of BR/EDR access code, channel access code (CAC), device access code (DAC) and inquiry access code (IAC). CAC is used for piconet identification while DAC and IAC is used for paging and inquiry respectively. The header, which is 54 bit long, is used for flow control, slave device addressing, error checking, as well as packet acknowledgement, numbering, and reordering where necessary. The payload, which can be from 0 to 2745 bit long, contains the payload header field and the payload body. The payload header field contains information used for routing the payload as well as the length of the payload body, while the payload body can carry both audio and data.

2.1.2 Bluetooth LE

LE radios can nominally support a data rate of up to 1 Mbps in most cases and 2 Mbps for uncoded data, achieved using GFSK [60]. It is worth noting that with error correction coding, the data rate is limited to either 500 kbps or 125 kbps for 2 symbols per bit and 8 symbols per bit, respectively. Similar to BR/EDR radios, LE radio also

operates in the unlicensed 2.4 GHz ISM band and also utilize FHSS to avoid destructive interference with other technologies occupying the same frequency band. Transceiver radio complexity is minimized with the use of shaped. binary modulation [16]. One of the key differences between LE and BR/EDR is that while the classic variant frequency hop between 79 channels each 1 Mhz apart with similar function (see Figure 2.5), BLE employs 40 channels each 2 MHz apart, of which 3 are specialized advertising channels (see Figure 2.8). BLE sytems, similar to BR/EDR cousins, hops between the 37 data channels in a pseudo-random pattern and can also avoid sections of the frequency band occupied by interfering devices. Similar to BR/EDR, LE also utilizes both FDMA and TDMA to partition its channels so that one frequency can be occupied by different LE devices at different time. Instead of time slots, the basic time unit for LE systems are called events, which are classified into advertising, extended advertising, periodic advertising, and connection events [16] [60].

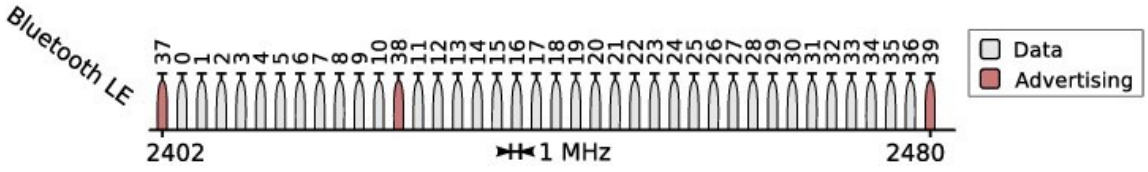


Figure 2.8: Bluetooth LE Channel Map. LE devices make use of 40 Channels, each 1 MHz apart, of which 37 are used to exchange data while the 3 advertising channels are used during the pairing process. [6]

For LE systems, there are three classifications of device types that participated in the events described above [16]. Advertisers are devices that transmit advertising packets on the three designated advertising channels. Scanners are devices that receive advertising packets but do not connect to the transmitting devices. Finally, Initiators are devices that receive advertising packets and connect to the transmitting devices. It is worth noting this classification only supplements the existing master/slave classification system previously observed in BR/EDR devices instead of replacing it. Specifically, advertiser/scanner/initiators designation are used during the linking process whereas the master/slave designations are used within the piconets after the link has already been established.

An advertising event is initiated when the advertiser starts the transmission of an advertising packet corresponding to the type of event on any given advertising channel. Once

the scanner receives the aforementioned advertising packet, it can choose to transmit a request to the advertisers on the same advertising channel depending on the type of advertising packet it receives. At this point, the advertiser may respond on the same advertising channel and the next advertising packet would be transmitted on a different channel. At any point during the advertising event, the advertiser may choose to end the event and the first channel used in the next advertising event would be the same first channel used in the previous advertising events [16] [60].

A connection event is preceded by an advertising event in which the initiator transmits a connection request to the advertiser on the same advertising channel that it receives the advertising packet. Within the connection request, the access address field is used to establish the channel hopping pattern. Channel hopping only occurs once at the very beginning for connection events [16]. Once the advertising event ends and the advertiser accepts the connection request, the connection event is initiated. As the connection is established, the advertiser becomes the slave, and the initiator becomes the master in the piconet. The master device has the ability to end the connection event at any point. The master device and the slave devices alternate transmission of data packets on the same data channel during the connection event. A device may be both a slave and master at the same time.

Generally, devices form physical links within a LE physical channel that provides bidirectional packet exchange between the master device and the slave devices [60]. Additionally, specialized physical links for advertising events provide unidirectional packet transport from the advertiser to a potentially unlimited number of scanners and initiators. As there might be multiple devices within one LE physical channel, the ability to create a physical link is restricted to a certain number of devices in several cases. Specifically, there must always be a physical link between a slave and its master, and slaves are permitted to have physical links to more than one master at a time but slaves within the same piconet do not form physical link.

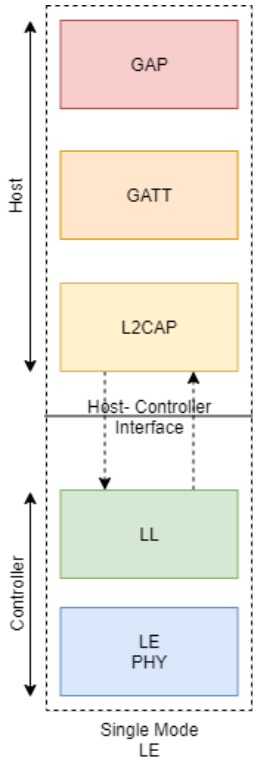


Figure 2.9: Example of a Typical LE Stack. The stack in this example follows the generic layered architecture with the LE upper layer consisting of the Generic Access Profile (GAP) and the Generic Attribute Profile (GATT).

Similar to BR/EDR, the LE stack follows the generic layered architecture (as shown in Figure 2.9) and can be partitioned into two main parts: the layers that inhabit the host, and those that inhabit the controller with the HCI acting as intermediary between the host and the controller [16] [62]. The Logical Layer for LE is called the Link Layer. The Link Layer and the Physical Layer both reside on the controller side while the Upper Layers reside on the host side. While the specific term for each of the layers of the Upper Layers are different, they generally fulfill the same function as that of the network, transport, session, presentation, and application layers in the OSI model [63].

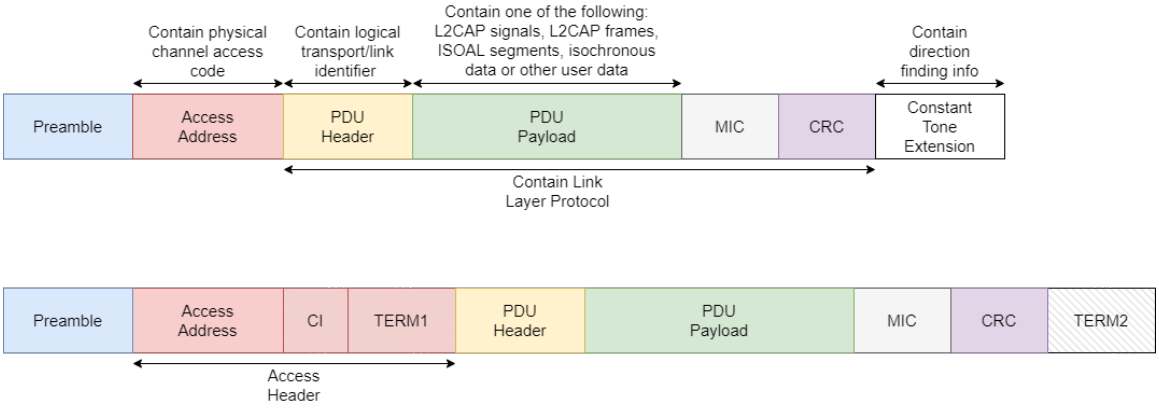


Figure 2.10: A LE uncoded packet alongside a LE coded packet. The main differences between uncoded and coded packet are the Coding Indicator (CI), Terminator 1 (TERM1), and Terminator 2 (TERM2) fields. While the CI field indicate the amount of symbols used in the coding scheme, the TERM fields form the terminator sequence needed to reset the encoder.

The generic LE packet consists of 4 main parts (See Figure 2.10): the preamble, the access address, the protocol data unit (PDU), and the cyclic redundancy check (CRC) [64] [65]. The preamble, 8 bit long in most cases and 16 bit long for uncoded data, is used for time and frequency synchronization as well as automatic gain control. Usually, the value contained within the preamble is set to fixed value, either 10101010 or 01010101 depending on the type of packet being sent as well as the least significant bit of the adjacent access address field [65]. The access address, which is 32 bit long, is used to distinguish communications on different physical channels operating within close proximity. Non-periodic advertising packet's access address for all types of advertising packets is set to the value 0x8E89BED6, while periodic advertising and data packet's access addresses are randomly generated. There are two main types of PDU: advertising channel PDU, and data channel PDU. Data channel PDU consists of a 16 or 24 bit header and payload field that can as long as 2040 bits, which also include an optional 32 bit message integrity check (MIC) field. Advertising channel PDU generally consists of 16 bit header field and a payload field that can be 0 to 296 bit long [65]. There are seven main types of advertising PDU: connectable un-directed advertising (ADV_IND), connectable directed advertising (ADV_DIRECT_IND), non-connectable un-directed advertising (ADV_NONCONN_IND), scannable un-directed advertising (ADV_SCAN_IND),

scanning request (SCAN _REQ), scanning response (SCAN_RSP), and connect request (CONNECT_REQ). These seven types are further organized into 3 main categories: advertising PDU, scanning PDU, and initiating PDU [64]. ADV_IND, ADV_DIRECT_IND, ADV_NON CONN_IND and ADV_SCAN_IND all belongs to the advertising PDU category while SCAN_REQ, SCAN_RSP belongs to the scanning PDU category. CONNECT_REQ is the sole member of the initating PDU category. The very last field within a LE packet is always the 24 bits long CRC field, used to detect errors in data storage and/or transmission. The CRC is calculated over the PDU using the polynomial $x^{24} + x^{10} + x^6 + x^4 + x^3 + x + 1$.

2.2 Packet Capturing Technologies

A packet capturing system, colloquially known as a packet analyzer or sniffer, is defined to be a system, which includes both hardware and software components, that can receive, intercept, and record passing traffic of a digital communication network [56] [58]. Originally conceived and used by network engineers to help diagnose and debug network issues, packet capturing systems are now utilized for a wide range of application in addition to their traditional role. Nowadays, aside from being utilized as security analysis and surveillance tools, modern packet capturing systems also serve as the basic building block for wireless testbeds. The Pal-6 packet capture system is shown as an example in figure 2.11.

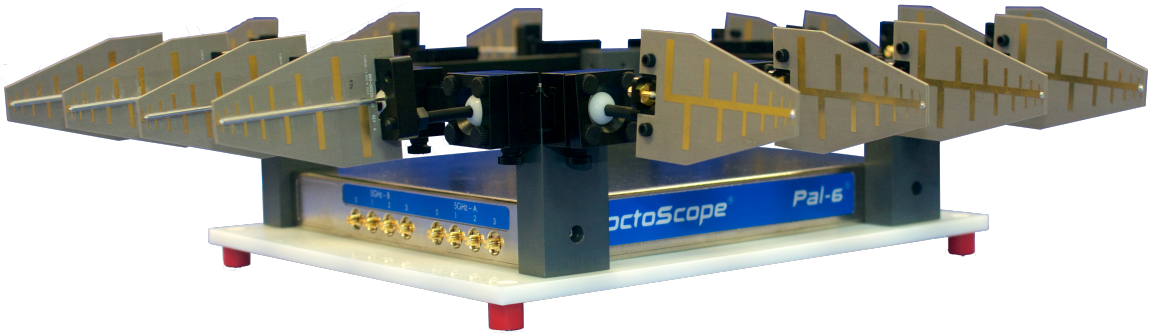


Figure 2.11: Example of a Packet Capture System. The octoScope Pal-6 Packet Capture System Complete With Antenna is Shown Here

A packet capturing system hardware component's main task is to enable the reception and interception of the monitored network's packet and traffic. Simply speaking, packet

capturing hardware components are usually just radios/transceivers designed for specific wireless communication technology but instead utilized in a sniffer system. Thus, the complexity of the hardware components varies significantly depending on the type of communication technology and sniffer in question. For instance, a hybrid Wi-Fi or an Ethernet sniffer can take advantage of built-in network interfaces on desktop computers to monitor Wi-Fi or Ethernet traffic whereas a Bluetooth sniffer may require a specialized transceiver in order to receive Bluetooth packets in promiscuous mode [1].

The software component of a packet capturing system's primary task is to control the hardware component, record and display the traffic captured in human-readable format, as well as decode the data if necessary. In contrast to the hardware components, the complexity for packet capturing software does not vary significantly and packet capturing software can generally be adapted to accommodate multiple communication technologies. Wireshark [66], previously known as Ethereal, is one of the most widely used and recognized sniffer software, and is a prime example for adaptability of the software components within a packet capturing system. An example of the Wireshark interface is shown in Figure 2.12. While it is originally intended to be use with Ethernet and Wi-Fi networks, Wireshark can also handle other communication standards as well with minor add-ons and modifications.

Generally, most sniffer systems are specifically designed and configured to monitor packets utilized by only one type of communication standard [57]. Thus, a sniffer system that is designed to monitor IEEE 802.11 Wi-Fi traffic cannot be guaranteed to be able to monitor Bluetooth traffic despite the fact that Wi-Fi and Bluetooth use the same 2.4 ISM frequency band. Although there are packet capturing systems that can be used to monitor with multiple types of packets utilized in different communication standards, these are subsequently more complex and expensive than systems that are designed for a single type of communication technology.

Overall, packet analyzers can generally be classified into three main classes of systems: (1) systems that only operates in inline mode, (2) those that only operates in promiscuous mode, and (3) those that can alternate between both inline and promiscuous mode [67].

[1] a device is said to be operating in promiscuous mode if it processes all received packets instead of only packets that is specifically addressed to be received by the said device

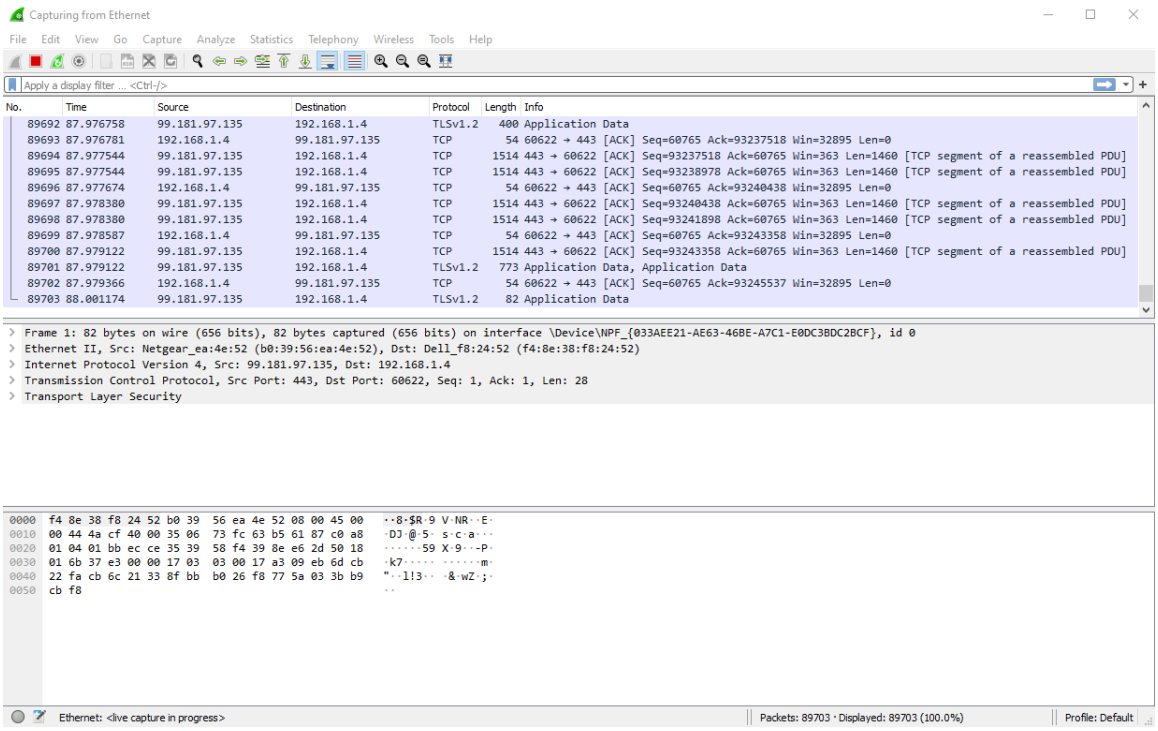


Figure 2.12: Picture showcasing the user interface of the Wireshark software. In this case, Wireshark is being utilized to capture Ethernet traffic

An inline sniffer refers to a device that that receives, transmits, and generally operates like a normal node within the same network that it is trying monitor. In contrast to the concept of inline sniffer, a promiscuous sniffer generally refers to a device that can listen in on and monitor traffic on a network without being a part of that network. In layman's terms, whereas the inline sniffers are "insiders", promiscuous sniffers are "eavesdroppers". While it is generally more straightforward to implement an inline packet capturing system, a promiscuous packet capturing system does offer the capability to monitor all the traffic traversing the network and may offer more information about the channels that the network operated in. It is worth noting there does exist a class of hybrid devices that combines the strength of both inline and promiscuous sniffers. Specifically, these so-called hybrid devices can seamlessly switch between operating in inline mode and promiscuous mode [56] [67]. By forcing the controller to pass on all the traffic that it receives to the host instead of sending only traffic that is specifically addressed to it, these hybrid devices can switch from operating

in inline mode to operating in promiscuous mode. Aside from being able to switch between inline and promiscuous mode, many modern packet analyzer systems also possess the ability to switch between unfiltered and filtered monitoring configuration [56]. In unfiltered mode, the sniffer will seek to capture and record all of the packets possible within its reach, whereas in filtered mode the sniffer will only capture packets that contain particular user-specified elements and parameters. Figure 2.13 illustrate the difference in the operating mechanism between inline and promiscuous sniffer. For a wired network, passing traffic could usually

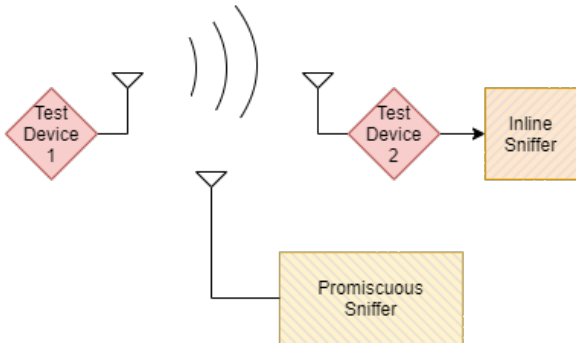


Figure 2.13: Comparison of inline packet capture system and promiscuous packet capture system's principle of operation

be captured through only one node in the network as there is no possibilities of packet being drop due to fading or interference from the environment [57]. Furthermore, equipment used to constructed the network normally have built-in tools to monitor the traffic themselves, removing the need of a third-party packet capturing system [67]. For wireless network, the task of intercepting traffic is much more daunting and complex as packet capturing systems need to take into account phenomena such as fading and interference in addition to other environmental variables that might affect the transmission and reception of a wireless signal. Moreover, in order to intercept traffic on a wireless network, a packet capturing system must first determined the frequency used by the wireless communication system to transmit and receive data. Such a challenge is especially true for a Bluetooth packet capturing system as the channel on which data packets is exchanged is consistently changed and switched around [16]. Overall, there are two main approach to determine the hopping pattern. The first method is to persistently monitor only one channel and log passing

target traffic to approximate the hopping pattern, whereas the second method is to monitor all the channels at the same time and outline the exact hopping pattern from observing the movement of the packets between the channels. While the first method is relatively straightforward and economical, it is limited by packet losses introduced by the amount of time needed to determine the hopping pattern as well as mismatch between the predicted and actual hopping patterns. The second method involves tracking multiple channels at once and recreating the hopping pattern through recording channel changes on the monitored channels. This approach is much more error-proof and much more complex and expensive to implement [26].

2.3 Packet Capture File Format

As previously mentioned, standardization of Bluetooth packet capture file format is still rather lacking as many manufacturers choose to come up with their own format than rather making use of pre-existing universal formats [26] [37]. However this trend is slowly reversing as more and more contemporary Bluetooth packet capture systems are designed to be compatible with more ubiquitous formats like BTSnoop, PCAP, and PCAPNG. [29] [27] [30]. The section below will give a brief description of the file formats utilized in the Bluetooth packet capture system presented in this thesis.

2.3.1 BTSNOOP

The original implementation of the sniffer system built in this project utilized HCI packets exchanged between the host PC and the controller on the sniffer hardware to document data exchanges within the Bluetooth piconet. Thus, in order to categorically log and display the HCI packets recorded, the BTSnoop file format was used. Originally created by Teledyne, the BTSnoop file format is also supported by many packet capturing software including Wireshark. While BTSnoop regularly serves as the standard file format for Bluetooth log files on Android devices [38], it is also suitable for usage in other devices. For example, no modifications were needed to adapt it to the requirements of our project [39].

A typical BTSnoop file, as shown in Figure 2.14, consist of one file header and multiple

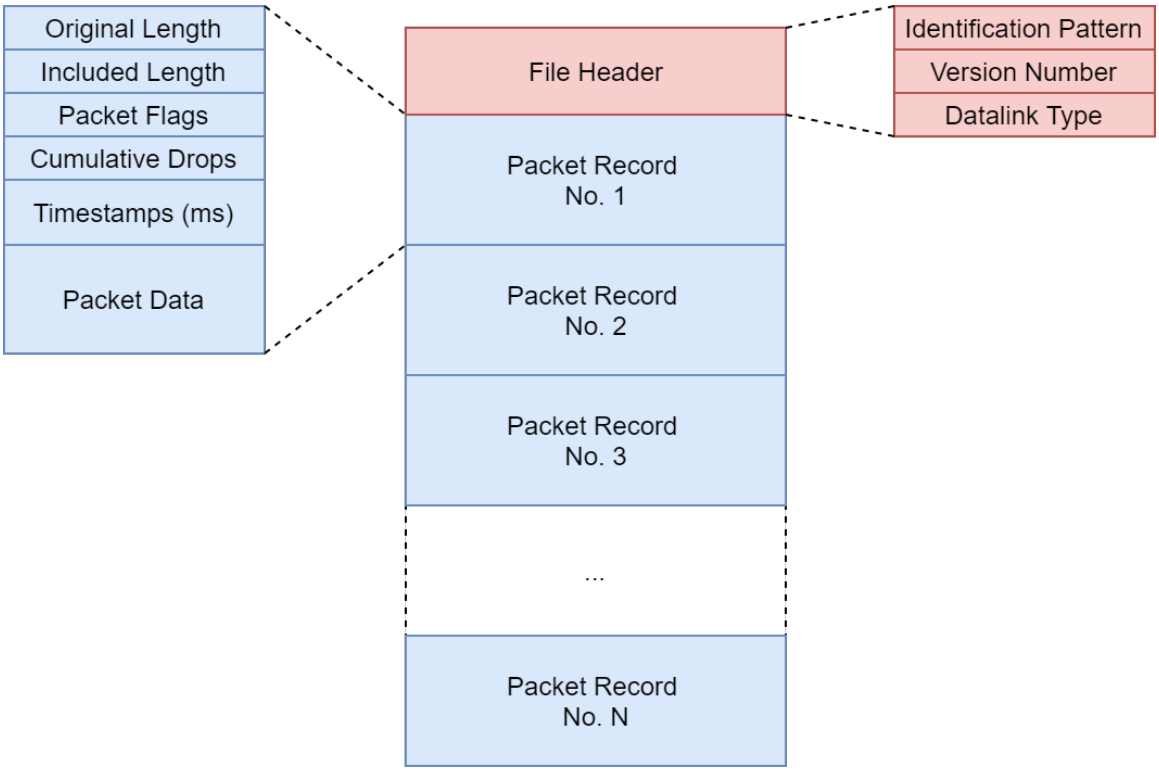


Figure 2.14: Overview of the BTSnoop Packet Capture File Format. In the BTSnoop format, one File Header is followed by multiple Packet Record, one for each packet captured

packet record field following the header [39]. The file header, which provides the format as well as some general information of the packet it contained, consists of three main fixed length components: the identification pattern, the version number, and the datalink type. The identification pattern, 64 bit or 8 octet long, is used to identify the file as a snoop packet capture file. The pattern used is always 62 74 73 6E 6F 70 00 in hex, which translates to BTSnoop followed by one null octet in ASCII. The version number, 32 bits or 4 octets long, is an unsigned integer value indicating the version of the packet capture file in question. In most cases, this field is generally set to 1 as a capture file is rarely reused in this project for any purposes. The datalink type is 32 bits or 4 octets long field that identify the type of datalink header used by the packet contained. Values 0 to 1000 are reserved whereas value 1001 and 1002 refer to un-encapsulated HCI H1 and HCI UART H4, respectively.

The packet record filed generally hold either a partial or complete copy a one packet in addition to some information regarding the packet it contained. In order to limit the file

size and the amount of packets contained within, the option to truncate the packet and only record part of it is available. The packet record is made up of two main section: information about the packet and the packet data itself. The information section is fixed at 24 octets long and is made up of five fields: original length, included length, packet flag, cumulative drops and timestamps. The original length and included length, both 32 bits/4 octet long, refers to the length of the packet contained before and after truncation. Generally, these two fields contain the same value as truncation rarely happens based on past experience. The packet flags field, also 32 bits/4 octet long, indicates the direction as well as the type of the packet contained. The cumulative drops field, 32 bits/4 octet long, shows the total amount of packets dropped by the system which created the packet file. The timestamp field, 64 bits/8 octet long, indicates the time of packet arrived since 12:00 AM January 1st, 0 AD nominal Gregorian in microseconds resolution. The packet data field is a variable length field that contain the captured packet including any necessary prefixes as required by the packet format. All integer value are mandated to be of big-endian order with the high-order bits first [39].

2.3.2 PCAP

The PCAP file format is one of the most basic and widely used formats used to record capture network data. Originally intended for use with libpcap [40], the PCAP format is supported by a wide variety of packet capture software, including Wireshark. In addition to Ethernet, the PCAP format can also accommodate other types of communication technology, including Bluetooth [40].

While there are some variations, the most common version of the PCAP format is version 2.4, which has not been changed since 1998 (see Figure 2.15). Generally, a PCAP file would contain one global header and multiple packet headers and packet data pairs for each of the packets that it contained [40]. The global header consists of 6 main fields: magic number, version major, version minor, timezone correction, timestamp accuracy, snapshot length, and data link type. The 32 bits/4 octet long magic number is used to identify the endianness of the file and to confirm whether the file is a PCAP file or not. The version major and minor field, each 16 bits/2 octet long, is used to indicate the version of the PCAP

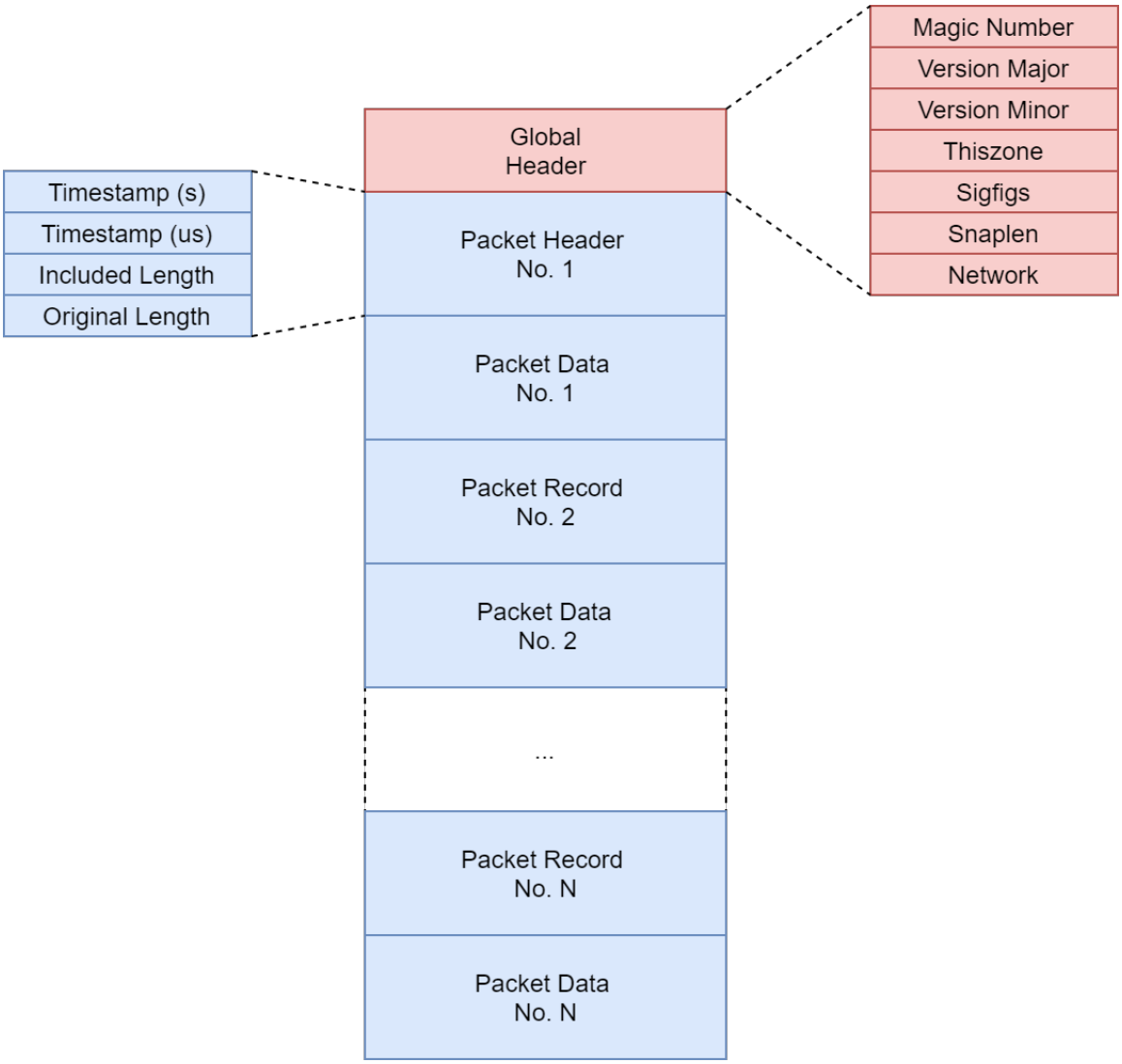


Figure 2.15: Overview of the PCAP packet capture file format. Similar to BTSnoop, the PCAP format consists of one Global Header and multiple Packet Header and Packet Data field for each packet logged

file and is always set to 2 and 4, respectively, for version 2.4, The timezone correction field, 32 bits/4 octets long, is used to indicate the difference in seconds between the Greenwich Mean Time/Coordinated Universal Time (GMT/UTC) and the local timezone. In practice, since the timestamp is always set in GMT, this field is always set to 0. The time stamp accuracy field, also 32 bits/4 octets long, indicates the number of significant figures in the timestamps. This value is usually set to 0 for the default option instead of 0 significant

figures in most cases. The snapshot length field, also 32 bits/4 octets long, indicates the maximum length of the captured packets in octets. The final field within the global header is the 32 bits/4 octets long data link type, which is used to indicate the type of communication technology of which packets are contained within the PCAP file [40]. The record header consist of four main 32 bit/ 4 octet sections: timestamp seconds, timestamp microseconds, included length and original length. The timestamp for the PCAP file are divided into two fields: one for seconds and the other for microseconds. Included length and original length refer to the length of the packet contained before and after truncation. It is worth noting that since truncation only occurs for very large packet sizes, original length and included length is the same value in most cases. The packet header is followed by the packet data, which is of variable length and has no specific data alignment.

2.3.3 PCAPNG

The PCAPNG file format is the continuation and improvement of the original PCAP format. Much like its predecessor, the PCAPNG is supported by a wide variety of packet capture software including Wireshark. Whereas the PCAP format can accommodate multiple communication technologies over different files, the PCAPNG format can accommodate multiple communication technologies on the same file. [41] [40]

The basic unit of PCAPNG file is the block (see Figure 2.16), and all blocks contain fields that identify the block type as well as the total length of the block in the beginning [41]. As the PCAPNG format is designed to be modular, there is no set order or requirement for the block included in a PCAPNG file. The sole exception to this rule is the Section Header Block (SHB), which must always be included at the beginning of the PCAPNG file and at the start of any section appended from one PCAPNG file to another. A PCAPNG configuration comparable to the classic PCAP configuration include one SHB in the beginning followed by one Interface Description Block (IDB), followed by multiple Enhanced Packet Block (EPB). The SHB, comparable to PCAP's global header, possesses general information such as the version number of the format, the total length of the packets, as well as the endianness of the data. The IDB is comparable to the packet header field in the PCAP format whereas the EPB is the equivalent of the packet data field. The IDB contains the snapshot length

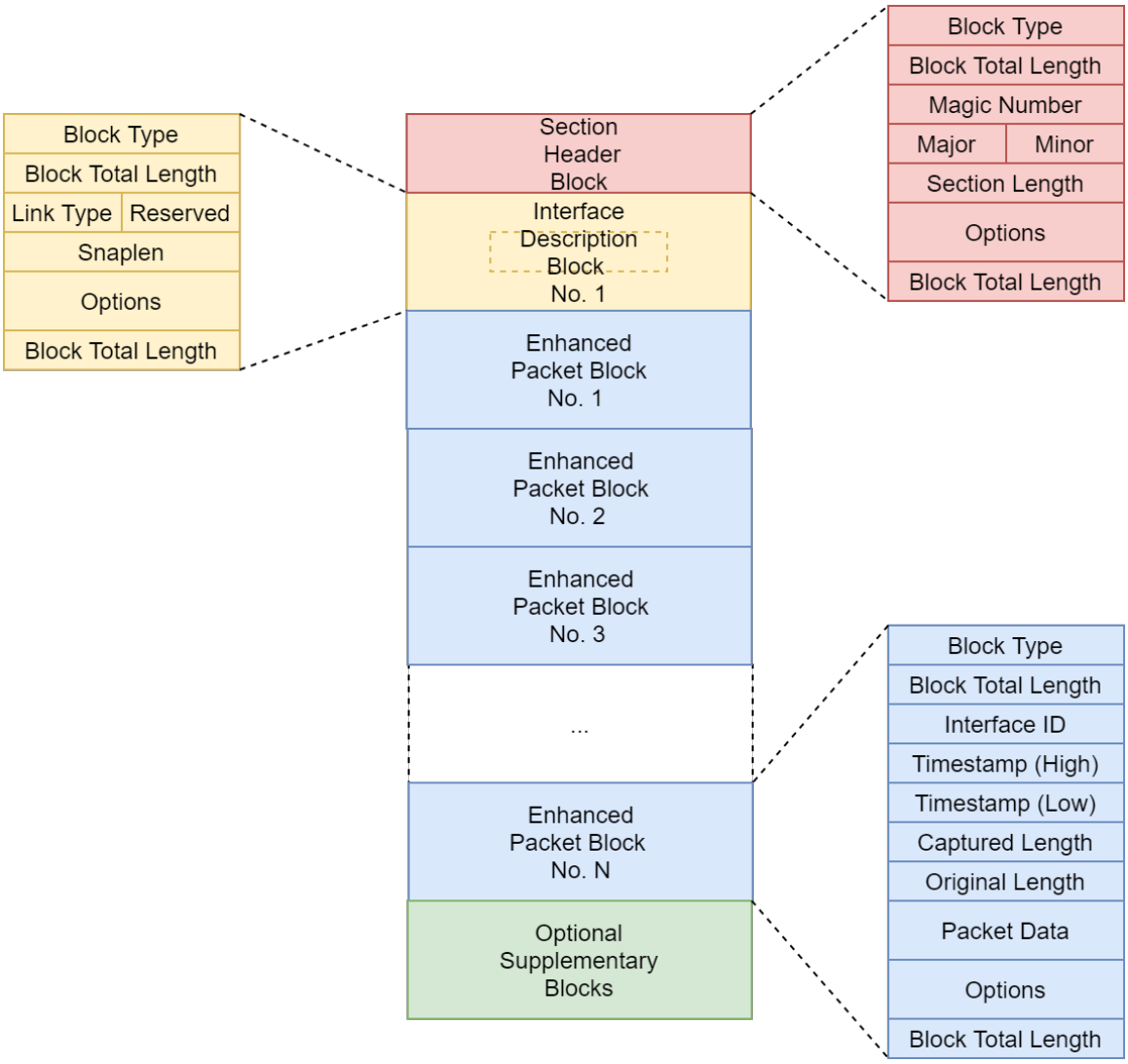


Figure 2.16: Overview of one typical variant of the PCAPNG packet capture file format. PCAPNG are designed to be modular and thus can be arranged in more than one way.

field as well as the data link field found in the global headers whereas EPB contain the interface ID, the timestamp as well as the included and original length of the contained packet [41].

2.4 Summary

In this chapter, background information regarding Bluetooth technologies, packet capturing systems, and data formats relevant to the project were presented. Specifically, Section 2.1 explored the general properties, the piconet, the architecture as well the packet format of both Bluetooth BR/EDR and BLE technology. Section 2.2 provide the general concept of packet capture systems, the hardware and software components that made up such systems, modes of operations, classification method as well as the implementation and limitation of contemporary Bluetooth sniffer systems. Finally, Section 2.3 laid out the details regarding the data format used to record

In the next chapter, a novel implementation of a testbed for Bluetooth technology is presented.

Chapter 3

Proposed Bluetooth Packet Logging and Formatting System

The work done over the last year and a half was a part of a larger effort (see [54]) to develop a wireless test-bed for technologies utilizing the 2.4 GHz band. Generally speaking, the effort's primary goal was to develop a system that could provide both accurate capture of packets as well as information regarding the RF environment through both inline and promiscuous packet capture, respectively, and then combine all the gathered data into a single packet capture file. Our part in the project was to develop hybrid promiscuous/inline packet capturing and logging capability specifically for both BR/EDR and BLE technology (see Figure 3.1).

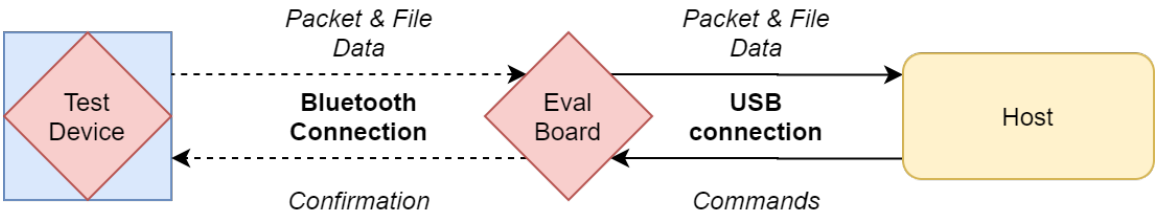


Figure 3.1: General architecture of the OPP (Object Push Profile) file transfer test system. The EVB will form a Bluetooth link with the test device but the packet and file data's final destination is the host.

This chapter pertains specifically to the development of the capability to indirectly

capture and log BR/EDR and BLE traffic into standard packet capture file formats such as BTSnoop, PCAP, and PCAPNG. Demonstration of the logging capability involves a file transfer between a BR/EDR and BLE capable devices and an EVB housing the CYW20719 Bluetooth IC (shown in Figure 3.2) [7], which is also connected through USB to a host computer. The system, which the logging function was implemented on, was developed and tested in [54] with assistance and guidance from engineers at octoScope and our advisor at WPI. A general description of this system, as well as technical data relevant to the packet capture capability, will be presented whereas specific technical detail as well as other functionality unrelated to the purpose of this thesis has already been presented in [54]. Overall, this part of the project successfully demonstrates the capability to log and translate packet captured through indirect inline sniffing to standard packet capture file format.

Figure 3.2: Overview of the EVB housing the CYW20719 Bluetooth IC used in this project [7] The EVB can be connected to a host computer using a micro-USB port shown on the left side. The built-in antenna is located on the daughter board on the right side

3.1 Test System Description

As briefly mentioned above, the test system used to demonstrate the logging and translating function consist of three main parts (see Figure 3.3): a BR/EDR and BLE capable deivce, a EVB housing the CYW20719 Bluetooth IC, and the host computer connected to the EVB through USB. The BR/EDR and BLE capable device, which in our test case is a smart phone running the Android operating system (OS), must first discover and then pair with the EVB (see Figure 3.4a). After being paired, a file transfer can be initiated from the BR/EDR and BLE capable device's side (see Figure 3.4b). At this point. the EVB must transmit a confirmation before the file transfer can begin. The host computer, the ultimate destination of the transferred file and also where the resulting packet capture file, is created and filled in. As there is not a way to extract the packet directly from the EVB, inline sniffing in a conventional sense is not possible. Instead, we rely on the HCI packets exchanged between the EVB, which is the controller, and the computer, which is the host, to recreate the conversation and data exchange between the two devices. Through this part of the project, we manage to successfully log packets into three types of packet capture file formats: the standard Bluetooth format BTSnoop as well as the more universal format PCAP and PCAPNG. The following sections below will present the work done in order to get each type of file format working.

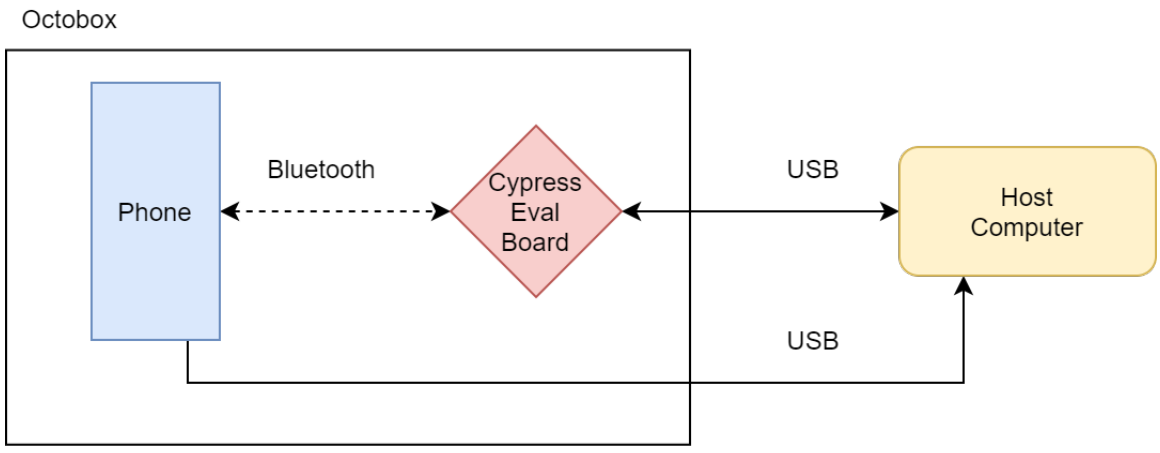
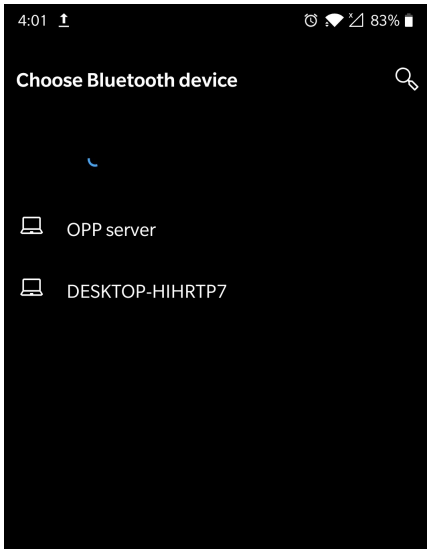
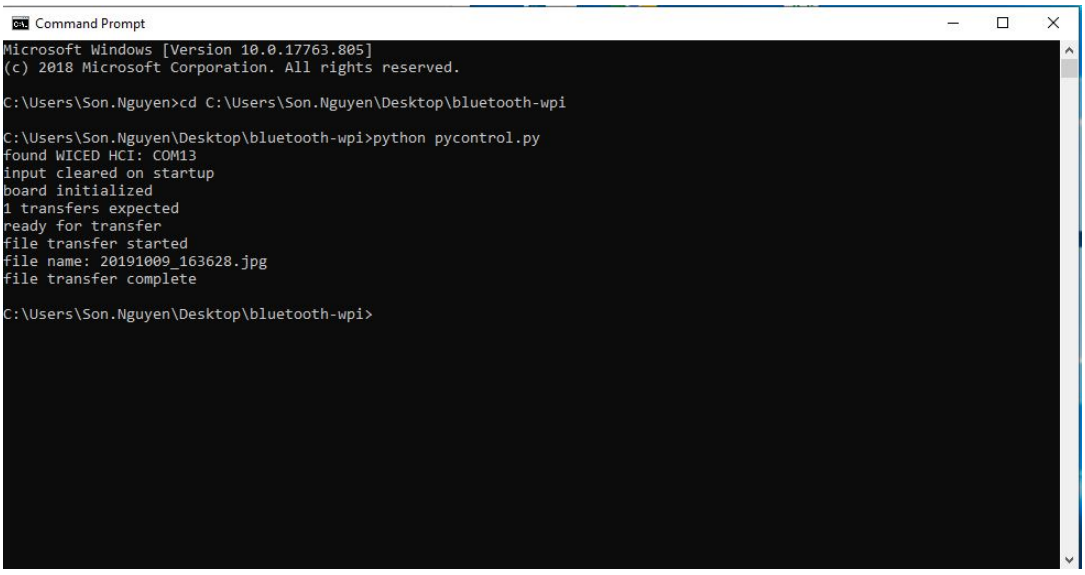


Figure 3.3: The test setup for the OPP file transfer test system. The test device used in this scenario is a smartphones running Android OS version 10. Both the phone and the EVB is connected to a host computer running Windows 10 through USB connections.



(a) Picture of the smartphone prompting the user to choose the destination for the file transfer



(b) Picture of a terminal window on the host PC as the file transfer is initiated and subsequently completed as the file and the capture log is received

Figure 3.4: Screenshot showing the file transfer on the phone and the host computer's side. The file transfer is started when the Python program is started on the host computer. At this point, the program waits for a file to be sent from the phone. On the phone's side, after selecting the file to be sent, the user is prompted to pick a destination. The EVB is designated as "OPP server" in this scenario. Upon successfully completing the file transfer, the EVB send both the transferred file and packet log to the host computer.

3.2 BTSnoop Formatting

While the original test system lacked the capability to log packets to a truly universal packet capture file format, it does possess the capability to log packets into the standard Bluetooth packet capture file format, BTSnoop. As all later logging function developed for more universal file format were built upon the original BTSnoop logging function, we felt it is necessary to include a description of it here.

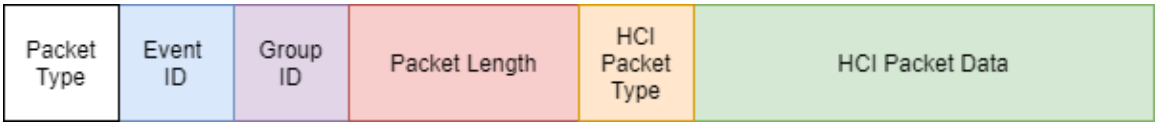


Figure 3.5: Illustration of the Cypress HCI packet. Out of the six fields, only the Packet Length field, HCI Packet Type field, and HCI Packet Data field is used in the formatting process.

While the file is transferred between the device, the EVB, and subsequently to its final destination at the host computer, the test program also consistently classify traffic that passes between the host and the controller, which in our case is the computer and the EVB, respectively. Overall, in our test case, there are four type of traffic that passes between the host and the controller: file transfer request, file transfer complete message, data packet and HCI packet (see Figure 3.5). While the purpose of the first two types of packets are pretty self-explanatory, data packets are used to recreate the transferred file on the host side and HCI packets are used to fill in the packet capture log. Thus, we are primarily interested in only the HCI packets here.

As the program classifies traffic passing between the host and the controller, HCI packets are specifically stored in an intermediate buffer for processing as all HCI packets are logged after and only after the file transfer is already completed. The data field as well as the command/direction field are kept separately for ease of processing and formatting as these two fields are placed in different and separated sections once the HCI packet they belong to is logged into the BTSnoop file.

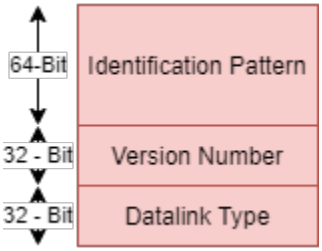


Figure 3.6: Illustration of the BTSnoop File Header field. The File Header field consist of the Identification Pattern, the Version Number and the Datalink Type.

Regarding the actual formatting of the BTSnoop log file, the format as detailed in documentation provided by Cypress and Teledyne were followed rigorously. The beginning of the file is always headed by a file header that contain three specific fields [39] [7] as shown in figure 3.6: the identification pattern, the version number, and the datalink type specifically. The identification pattern is a 64-bit pattern, which can be translated to the ASCII string "BTSnoop" followed by one null octet, that identify the file as a BTSnoop file. The version number is a 32-bit unsigned integer value indicating the version of the BTSnoop file in question. Based on the documentation provided by Cypress and Teledyne, there has not been any update or changes after its initial release, the version number is always set to the value 1. The datalink type field is a 32-bit long value and is used to identify the datalink header employed in the packet logged in the BTSnoop file in question. For simplicity sake, we initially choose to utilize un-encapsulated HCI H1 packets for host-controller communication so the system always sets this value to 1001 (03E9 in hex) in our tests.

The file header is followed by a number of packet records as shown in figure 3.7. The number of packet records is equal to the number of packets that are captured by the BTSnoop file. For each HCI packet logged, the first two fields in a packet record are reserved for the number of bytes in the original packet as well as the number of bytes actually included in the BTSnoop file, respectively. The reason there are two separate fields and the distinction exists is that the packet might be truncated once the number of bytes in a given packet exceeds the upper limit imposed by the packet capture program and thus would cause the number of bytes included and the number of bytes in the original packet to be

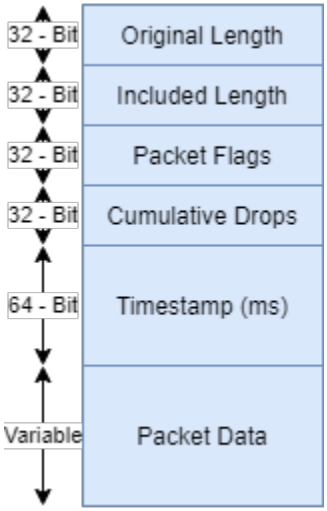


Figure 3.7: Illustration of the BTSnoop Packet Record field. The Packet Record field consist of the Original Length, the Included Length, the Packet Flags, the Cumulative Drops, the Timestamp, and the Packet Data

different. In reality, these two length fields are almost always of the same value as none of the packets logged in any of the tests we conducted were ever truncated.

The 32-bit packet flags field, the third field in a packet record, is used indicate the direction and command flag of the packet. This field is filled in with the command/direction field previously separated in the classification process. It is worth noting that due to differences in the way the direction and command fields are encoded in the Cypress packet, these two fields need to be converted before they can be used to fill in packet records. Specifically, while the BTSnoop file's direction flag indicate "sent" for value 0 and "received" for value 1, the opposite is true for Cypress packets. Similarly, the encoding for the command flag is also flipped in this manner. Thus, we have to take the value given in the Cypress packet and subtract it by 3, effectively bit-flipping the direction and command flag bits, to convert it to the proper value used by the BTSnoop format. The fourth field in a packet record, the cumulative drops field, is a 32-bit unsigned integer indicating the amount of packets that were dropped by the packet capture system between the first packet and the packet in question. As noted in documentation provided by Cypress and Teledyne, it is often not necessary to fill in this field at all if the test system lacks the ability to count dropped

packets. Thus, for simplicity and ease of implementation, we decided to not include this function in our test system. The timestamp field is the fifth and penultimate field within a

Algorithm 1 Generating microseconds resolution timestamp compliant with BTSnoop format

1: Initialize $t_{btsnoop} = t_{epoch} = temp = 0$
2: Generate epoch timestamp using native epoch time function
3: t_{epoch} = generated epoch timestamp
4: $temp = t_{epoch} * 1000000$ ▷ Converting from seconds to microseconds
5: $t_{btsnoop} = temp + 66463223296000000$ ▷ Adding arbitrary offset value
6: **Return** $t_{btsnoop}$

packet record. As the name suggested, this field is filled with a 64-bit signed integer that indicate the packet's arrival time in microseconds since midnight, January 1st, 0 AD nominal Gregorian. It is worth noting that the BTSnoop files make use of its own the timestamp format instead of the standard Unix time format utilized in many other packet capture format. Furthermore, since built-in time functions for most programming language make use of the standard epoch time, a somewhat complicated conversion is necessary. Simply adding the difference between Unix epoch time and the non-standard time format to the result of the Python time function yields the timestamp that are inaccurate and varied quite wildly. In [54], it was discovered through extensive testing that by adding the equivalent of over 2100 years in microseconds to the result, an accurate timestamp can be obtained (see Algorithm 1). Since BTSnoop was but a stepping stone, we never dug deeper into figuring out why this specific arbitrary value worked. After the timestamp, the actual packet data is the final field within a packet record. This field was filled in a rather straight-forward manner without any significant modifications or conversion. Figure 3.8 showcased a successful BTSnoop capture.

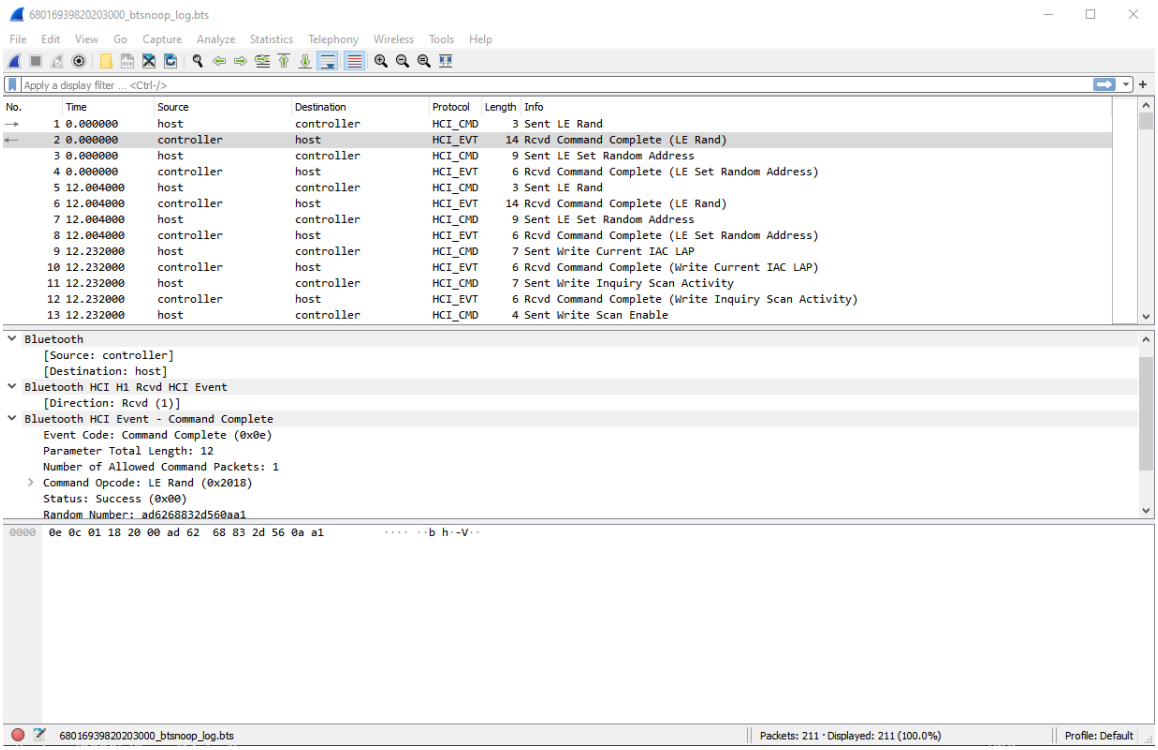


Figure 3.8: Example of a successful capture logged into a BTSnoop File. Note that the BTSnoop file was able to indicate both the direction and the classification of the packets logged.

3.3 PCAP Formatting

With the test system's successful demonstration of the capability to log packets to the standard Bluetooth packet capture file format, we next move on to the more universal PCAP packet capture file format. While BTSnoop is only compatible with Bluetooth technologies such as BR/EDR and BLE, PCAP can be used with a wide variety of communication technologies including Bluetooth, Wi-Fi, and Ethernet among others. Overall there are many similarities between the new system and its previous iteration. The new test system also classifies traffic that passes between the host and the controller, which in our case is again the computer and the EVB, respectively. Traffic between the host and controller are similarly classify into four types: file transfer request, file transfer complete message, data packet, and HCI packet. HCI packets, the main priority for the logging function, are again specifically stored in an intermediate buffer and they are only logged after the file transfer

is already completed. The main differences between the two systems occur with respect to the formatting and logging procedures for BTSnoop and PCAP. Overall, there are three main parts in a PCAP file: the global header for the entire file, as well as paired packet header, and packet data fields for each packet logged.

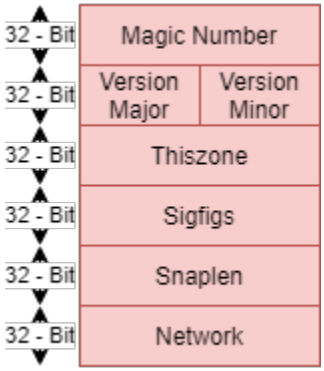


Figure 3.9: Illustration of PCAP Global Header field. The Global Header consist of the Magic Number field, the Version Major/Minor field, the Thiszone field, the Sigfigs field, the Snaplen field, and the Snaplen field.

The global header (see Figure 3.9) begins with a 32-bit magic number field that basically serves to identify the file as a PCAP file, similar in purpose to BTSnoop's identification pattern. As opposed to translating to a certain ASCII string, PCAP's magic number value is simply the unsigned integer that is 0xa1b2c3d4 in hex. The dual 16-bit version major and minor field is used to indicate the version of the file in question. The test system makes use of version 2.4, the latest version as outlined in official documentation provided by Wireshark. The 32-bit thiszone field is intended to store the correction time value in seconds between GMT/UTC and the local timezone. This is most likely a redundancy mechanism as timestamps are always in GMT in practice. In the PCAP test system, the correction time value in thiszone is always set to 0. The 32-bit sigfigs field indicates the accuracy of the timestamps in the number significant figures included in any given timestamp. To avoid any unecessary complications, the system always set this field to 0 as to select the default option for the number of significant figures in a timestamp.

The final two 32-bit field within the global header is the snaplen and the network field. The snaplen field, short for snapshot length, sets the upper limit of the number of bytes of

a packet data captured that can be logged before it is truncated. In order to minimize the amount of information lost, we set the snaplen to be the maximum value 65535, or 0xFFFF in hex. The network field specifies a link-layer header type at the beginning of the packet. Although our previous system utilizes only HCI H1 for simplicity, PCAP file only supports HCI UART H4 and HCI UART H4 with PHDR direction field. Thus, the value for the network field can only either be set to 187 (0xBB) for HCI UART H4 or 201 (0xC9) for HCI UART H4 with PHDR.

Algorithm 2 Generating microseconds resolution timestamp compliant with PCAP format

1: Initialize $t_{pcap\_s} = t_{pcap\_us} = t_{epoch} = 0$
2: Generate epoch timestamp using native epoch time function
3: t_{epoch} = generated epoch timestamp
4: $t_{pcap\_s} = int(t_{epoch})$ ▷ Cast to int to remove decimal values
5: $t_{pcap\_us} = (t_{epoch} - t_{pcap\_s}) * 1000000$ ▷ Extracting and converting microseconds offset
6: **Return** t_{pcap_s}, t_{pcap_us}

For each packet logged to a PCAP file, there is a record (packet) header that contains the timestamps and the lengths of the aforementioned packet (see Figure 3.10). Overall there are four 32-bit fields in a record (packet) header: ts_sec, ts_usec, incl_len, and orig_len. The ts_sec field, utilizing the standard Unix time format, contains the date and time when the packet was captured in seconds since January 1, 1970 00:00:00 GMT. The ts_usec is often used to contain the microseconds offset to ts_sec for regular PCAP but it can also be used to contain the offset in nanoseconds for a nanosecond PCAP file. As the Unix time function in Python is limited to microseconds resolution, the PCAP test system is also limited to microseconds offset. Since the PCAP timestamp makes use of Unix time, the PCAP system does not have any issue with the timestamp relative to the BTSnoop system as it is relatively straightforward to obtain the ts_sec and ts_usec. Ts_sec can be obtained by taking only the integer portion of the result of the Python Unix time function whereas ts_usec can be calculated by subtracting ts_usec from the unmodified result of the Python unix time function (see Algorithm 2).

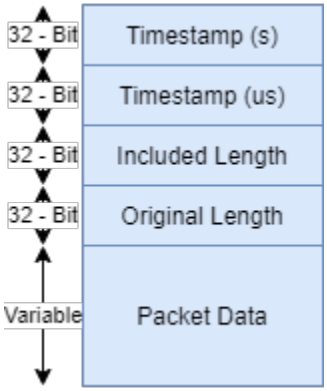


Figure 3.10: PCAP Packet Record field. The Packet Record consist of both the seconds and microseconds component of the Timestamp, the Included and Orginal Length, and finally the Packet Data.

The incl_len and the orig_len field, as their name suggested, contains the number of bytes of the packet data actually logged into the PCAP file and the original number of bytes of the packet, respectively. In practice, these two fields always contain the same value as the PCAP system has its snaplen value set to the highest value possible. As the two length fields round out the record (packet) header, the actual packet data immediately follows after. At first, due to a lack of documentation detailing the differences between HCI H1 and H4 packet, it was assumed that the HCI H4 could substitute for the HCI H1 header without any changes and modifications to how individual packets are processed and logged. In practice, logging packets to a PCAP file without any modifications would result in a unspecified packet type error. The issue was not limited to only HCI H4 headers but also HCI H4 PHDR headers as well. In fact, switching to HCI H4 PHDR complicates the issue even further as resulting PCAP files would be plagued by not only unspecified packet type errors but also unspecified direction errors as well. This still produced several useful outcomes, as the HCI H4 PHDR tests showed that the issue clearly lies in how the packet was process before it was logged into the PCAP file. Unsuch as BTSnoop, PCAP files do not possess their own direction and command flag field but instead rely on the data packet's own direction and classification. Thus, the previously separated direction and command field must be appended to the packet data before both are logged into a PCAP file. While

this solution completely resolve the unspecified direction error. it is only a partial solution to the unspecified packet type error. Specifically, even though around half of the packets were recognised by Wireshark, the unspecified packet type error curiously persist in the other half. Without a thorough solution in sight, and thinking that a newer packet capture file format could solve the issue, the project was moved to its next stage, adapting the system to the PCAPNG format. Figure 3.11 showcase a successful PCAP capture.

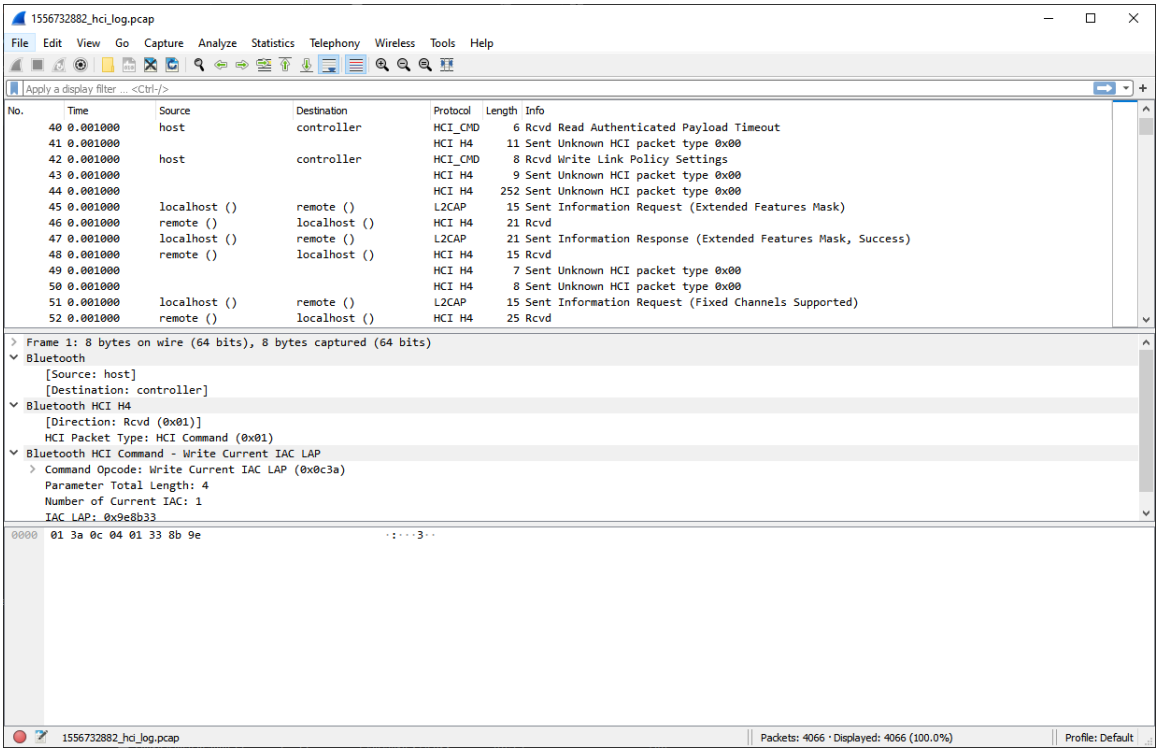


Figure 3.11: Example of a successful capture logged into a PCAP file. Note for some of the packets, the direction field and the packet type field is unfilled due to the an encapsulation issue discovered later on.

3.4 PCAPNG Formatting

As its name suggested, PCAPNG is the evolution of the PCAP file format. PCAPNG is intended to improve upon PCAP's many drawback including the lack of standard nanosecond resolution, interface information and packet drop count among other less notable issues. For this project, it is a first step forward as PCAPNG can be considered a truly universal platform. Specifically, PCAPNG allows Bluetooth packets to be logged concurrently with packets utilizing other technologies whereas PCAP only offers the ability to append a Bluetooth packet capture file to a packet capture file utilizing a different technology. Furthermore, it was hoped that a newer and more updated platform could be the key to solve the persistent unidentified packet type error.

While the two file formats do share certain similarities, PCAP and PCAPNG files are generally structured very differently. PCAPNG files, designed with modularity in mind, can be arranged in more than one way, whereas a PCAP file can only be arranged in one. To be more specific, a PCAPNG file is made up of a basic data unit called blocks. The general block structure consist of a block type field, a block total length field, a block body field, and another block total length field, respectively. The block type and the two blocks total length field are 32 bits long while the block body field are of variable length but is padded to the closest multiple of 32 bits.

There are a wide variety of blocks that are classified into two categories: mandatory blocks, which must appear at least once in a PCAPNG file, and optional blocks, which may or may not appear. The official documentation mandates that the only mandatory blocks are the Section Header Block (SHB) whereas Interface Description Block (IDB), Enhanced Packet Block (EPB) Simple Packet Block (SPB), Name Resolution Block (NRB), Interface Statistics Block (ISB), and other custom vendor-specific block are optional blocks. It is worth noting that there are blocks mentioned within the official PCAPNG documentation that falls into neither categories, only being labeled as experimental as they are being tested. To avoid unnecessary complications, the new test system is strictly incompatible with any experimental blocks.

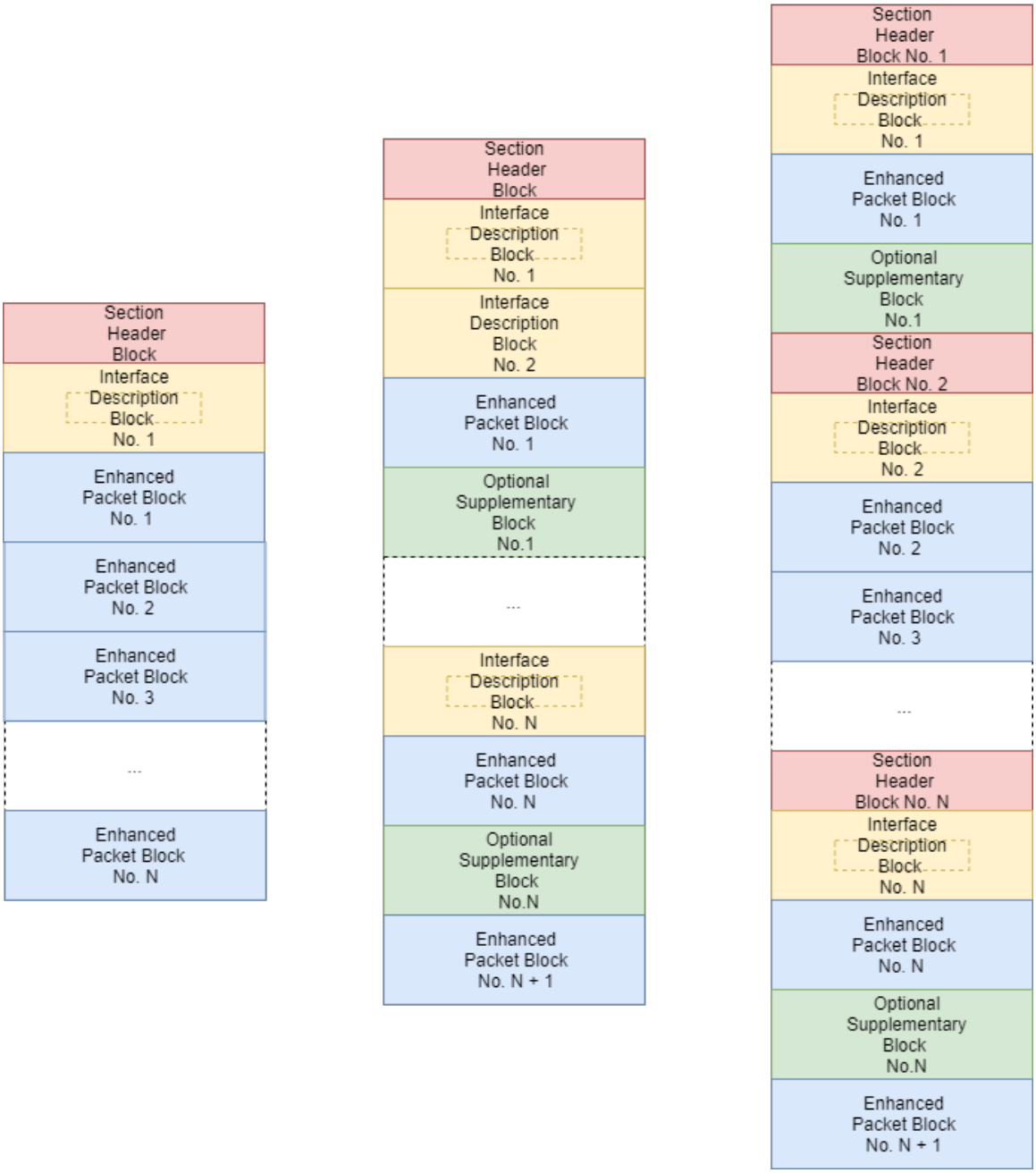


Figure 3.12: Examples of How PCAPNG Files Maybe Configured by Rearranging Different Types of Blocks

A PCAPNG file resembling a classic PCAP file, which represents the bare minimum for a useful capture file, would consist of one SHB, one IDB, and a few EPBs. More complex PCAPNG files, as shown in figure 3.12, may consist of multiple SHBs, IDBs, and EPBs, as

well as complementary NRBs and ISBs. Ultimately, a more complex format would allow packets utilized in different protocols, technologies, and standards to be logged into the same PCAPNG file. However, the simpler format, is already adequate to demonstrate the capability to log Bluetooth packets to PCAPNG while also being easier to implement.

For all types of format, a PCAPNG file is always started with a SHB (see Figure 3.13) followed by an IDB. The SHB and IDB is generally considered the equivalence of the global header in PCAP file, providing general information about the capture file. In the simple format intended for use, data blocks such as the SPB and the EPB immediately follow after the SHB and IDB. The SPB is similar to the EPB but it is lighter and simpler to process as it only contain a minimal set of information. While a PCAPNG file can accommodate both types of packet block, it is decided that the test system will only log packets to EPB for simplicity sake. The SHB, IDB, and EPB all follow the standard block structure, only differing in the structure of block body and the content of the block type field.

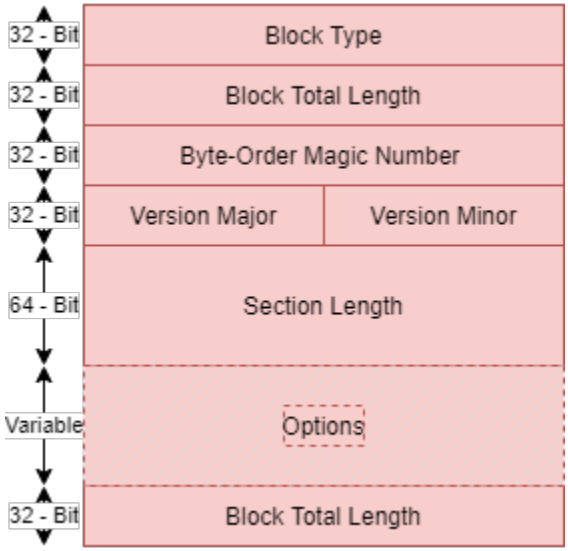


Figure 3.13: Illustration of PCAPNG Section Header Block (SHB). The SHB follows the generic block structure and contains the Byte-Order Magic Number field, the Version Major/Minor field, and the Section Length field.

The SHB block type value is 0x0A0D0D0A in hex, which is unique as all other block type value is a single number. The SHB's block body consist of the byte-order magic field,

the major version field, the minor version field, the section length field and the options field. The 32-bit byte-order magic field store the unsigned magic number 0x1A2B3C4D which help identify PCAPNG file and indicate the endianness of each section. Similar to their PCAP counterpart, the two 16-bit major version and minor version field indicate the current version of the PCAPNG file at issue. Currently, the system is making use of version 1.0, which, as of August 9th 2020, is still the most recent version as indicated by official documentation. The 64-bit section length contains a signed value specifying the length in octets of the following section, which can be used to skip the section. Alternatively, the section length value can be set to 0xFFFFFFFFFFFFFFFF to indicate that the length of the section is unspecified at the cost of being unable to skip the section. Since there is no use for a skip function, this value is set to 0xFFFFFFFFFFFFFFFF in the system. The variable length option field can be used to enable a number of optional settings; this field is excluded for none of the settings is used in the test system.

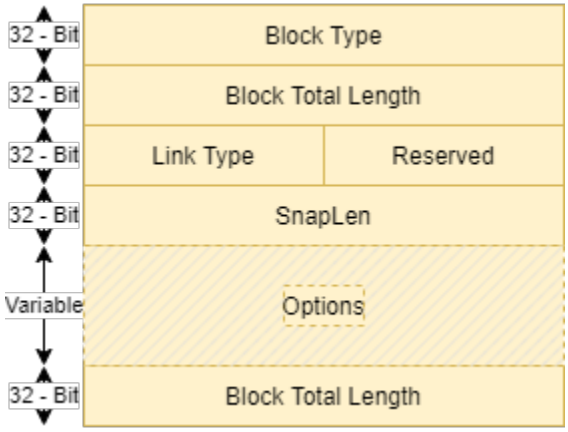


Figure 3.14: Illustration of PCAPNG Interface Description Block (IDB). The IDB follows the generic block structure and contains the Link Type field, and the SnapLen field. There is also a 16-bit field that is reserved for future use that is currently not utilized for any purpose.

The IDB block type value is 0x00000001 in hex. As shown in figure 3.14, the IDB's block body consist of the linktype field, the snaplen field and the options field. The 16-bit linktype field contains an unsigned value that indicates the link layer type of the interface. Semantics aside, this field fulfills the same purpose as the network field for PCAP files and

thus make use of the same value table to distinguish between different link layer type. As such, the linktype field can also be set to 0xC9 to indicate HCI H4 with PHDR packet type. It is also worth noting that a 16-bit reserved field, which is kept specifically for future use and always filled with 0, immediately follows the link type field. The third field within the IDB's block body is the SnapLen field, similar to its counterpart in the PCAP file, contains the value that is the upper limit of the number of bytes of a packet data captured that can be logged before it is truncated. In the PCAPNG test system, the SnapLen value is set to 0, indicating no upper limit. The variable length option field in IDB is not utilized and therefore omitted.

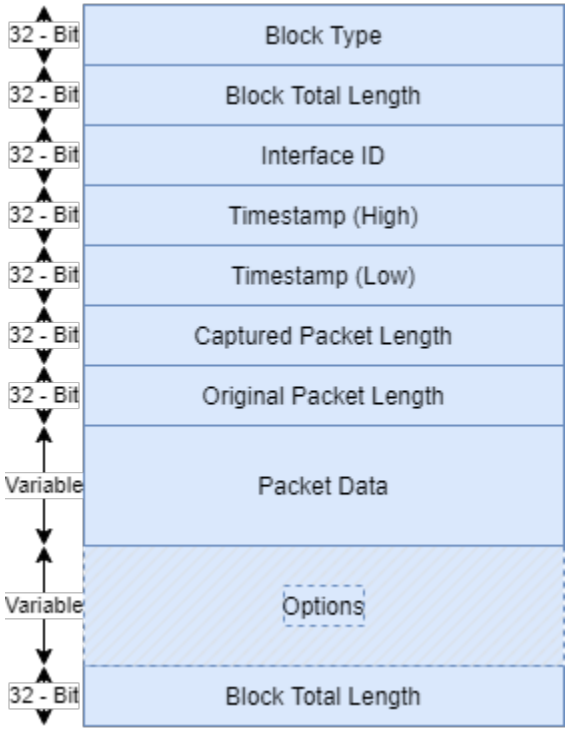


Figure 3.15: Illustration of PCAPNG Enhanced Packet Block (EPB). The EPB follows the generic block structure and contains the Interface ID field, the Timestamp, the Captured/Original Packet Length fields, the Packet Data field.

The EPB block type value is 0x00000006 in hex. As shown in figure 3.15, the EPB's block body consists of the interface id field, two complementary timestamp fields, captured/original length fields and the packet data field. The test system also does not make

use of the option field in the EPB. The 32-bit interface ID field contain an unsigned value that specify the interface on which the packet logged was either transmitted or received. It should be noted the interface ID value is the same as the index of the corresponding IDB, which is 0 in this case. The two 32-bit timestamp fields store the upper and lower 32-bits of a 64-bit timestamp in Unix time format. In order to obtain the 64-bit timestamp, the result of the Python time function was multiplied to one million. The upper half of the timestamp is obtained by bit-shifting the 64-bit timestamp by 32 while lower half is obtained by adding the 64-bit timestamp with 0x00000000ffffffff (see Algorithm 3).

Algorithm 3 Generating microseconds resolution timestamp compliant with PCAPNG format

1: Initialize $t_{high} = t_{low} = t_{epoch} = t_{us}0$
2: Generate epoch timestamp using native epoch time function
3: t_{epoch} = generated epoch timestamp
4: $t_{us} = t_{epoch} * 1000000$ ▷ Convert epoch timestamp to microseconds
5: $t_{high} = int(t_{us}) >> 32$ ▷ Extracting the higher 32 bits of the microseconds epoch timestamp
6: $t_{low} = int(t_{us})$ & 0x00000000FFFFFFFF ▷ Extracting the lower 32 bits of the microseconds epoch timestamp
7: **Return** t_{high}, t_{low}

The 32-bit captured packet length and original packet length fields fulfill the same purpose as their PCAP counterparts, indicating the number of bytes actually logged into the PCAPNG file and the original number of bytes of the packet, respectively. The packet data field is the penultimate field within the block body of a EPB. Due to its nature, the packet data field is always of variable length. The length of the packet data field is always a multiple of 32, since the field is padded to a 32-bit boundary to comply with the standard block size. The number of bytes to be padded is calculated by the standard padding formula as laid out below, with Padding is the amount of bytes that needs to added, Align is the target number of byte for the padding operation and Offset is the differences between the original number of byte and the target.

$$\text{Padding} = \text{Align} - [(\text{Offset})\%\text{Align}]\%\text{Align} \tag{3.1}$$

As previously mentioned, it was hoped that by switching to the updated PCAPNG format, the unidentified packet issue may be resolved. It was soon proven the issue does not lie in the file format but rather something deeper within the system since the same error was still being encountered. While the switch did not fix the issue, it did narrow down the suspicion to one component, namely the Cypress EVB. Our suspicion was confirmed when we discovered, despite there being four types of HCI H4 packets, the Cypress EVB only label two types. Specifically, the Cypress EVB only label EVT and CMD packets while SCO and ACL packets were ignored. Even with this information, the solution was no closer to being discovered as there was still no method to distinguish between SCO and ACL packets. Upon closer inspection of the Bluetooth core specifications, it was realized the data total length field for the SCO and ACL packets can be used to distinguish between the two types of packets. To be more specific, while the SCO's data total length field is suppose to be only 8 bit, its ACL counterpart is 16 bit long. Thus by checking whether bit 16 to 24 or bit 16 to 31 matches the total length of the packet, we can distinguish between SCO and ACL packet (see Algorithm 4). Figure 3.16 showcase a successful PCAPNG capture.

Algorithm 4 Identify and Distinguish HCI SCO and ACL packets

```
1:  if packet_type != EVT or CMD then
2:      ACL_len = packet[16 : 31]
3:      SCO_len = packet[16 : 24]
4:      if ACL_len == len(packet) then
5:          packet_type = ACL
6:      else if SCO_len == len(packet) then
7:          packet_type = SCO
8:      else
9:          packet_type = unidentified
10:     end if
11: end if
```

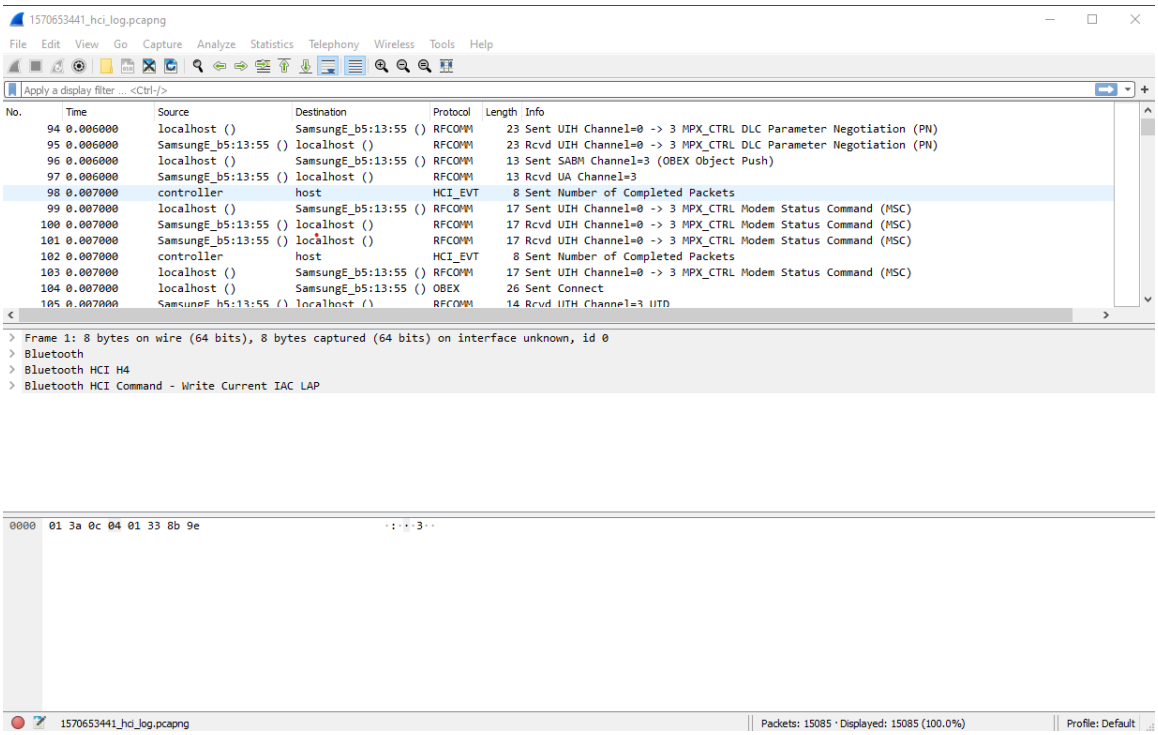


Figure 3.16: Example of a successful capture logged into a PCAPNG file. Note that the direction and the packet type field are once again filled in for all packets.

3.5 Summary

In this chapter, relevant technical details, design choices, and challenges of each of the three system were presented. The chapter was ordered according to the development timeline and follows the project progression through increasingly complex and universal packet capture file format. Section 3.1 details the development of the standard Bluetooth packet capture, BTSnoop. Subsequently, Sections 3.2 and 3.3 also follows the development of PCAP and PCAPNG file format, respectively. Overall, while the process experienced many difficulties and challenges at various stages of the development process, the capability to log Bluetooth packets into all three format was successfully demonstrated.

In the next chapter, the feasibility and development process of a hybrid inline/promiscuous packet capture system is explored and presented.

Chapter 4

Proposed Bluetooth Packet Capturing System

As mentioned briefly at the beginning of Chapter 3, this thesis presents the work done as part of larger effort to develop a a wireless test-bed for technologies utilizing the 2.4 GHz band. Generally, the system was supposed to provide both accurate capture of packets as well as information regarding the RF environment through both inline and promiscuous packet capture, respectively, and then combine all the gathered data into a single packet capture file. With the inline component already developed (see [54]) and development of logging/formatting subsystem also wrapping up, the final missing part of the project is the promiscuous packet capture system.

Despite the availability of promiscuous Bluetooth sniffers, it was decided the project will attempt to built a promiscuous packet capture system based on the Litepoint SDR. This is because of the limitations of commercially-available Bluetooth ICs as well as the capability of the Litepoint SDR. To be more specific, most affordable commercially-available Bluetooth sniffers are limited to monitoring only one Bluetooth data link, while systems that possess the ability to monitor multiple link at once are prohibitively expensive and inaccessible. The Litepoint SDR, which was already used by octoScope for production testing, not only offers the ability to monitor multiple Bluetooth data link at once but also provide additional information of RF environment. Attempts were made to build the

promiscuous sniffing subsystem around the Litepoint IQxel-M16W [8] at first, but then the IQxel-MW 7G [9] was utilized in the final build of the subsystem instead. This chapter will first seek to explore the technical details of both version of the promiscuous packet capturing subsystem, as well as explain how the subsystem is suppose to fit into the larger hybrid system.

4.1 Litepoint IQxel-M16W-Based Promiscuous Packet Capturing Subsystem

Unlike as the CYW20719, the Litepoint IQxel-M16W (see Figure 4.1) was originally designed to be a dedicated piece of test equipment for multiple types of communication protocols and technologies instead of a dedicated Bluetooth IC. Thus, building the promiscuous sniffing function around the IQxel-M16W offers both unique advantage and challenges. The built-in web interface is a good example to illustrate this dilemma. While the web interface makes it easier and more intuitive to interact with IQxel-M16W, almost every option included within the web interface needed to be carefully fine-tuned in order to even begin logging Bluetooth packets.

Figure 4.1: The Litepoint IQxel-M16W Used in the First Build of the Promiscuous Packet Capture System [8]. The ten status lights to the left indicate the on-off state of device, whether a session is active, and the status of each of the 8 modules. Each module consist of two ports that can be configured as an input or an output and monitored separately.

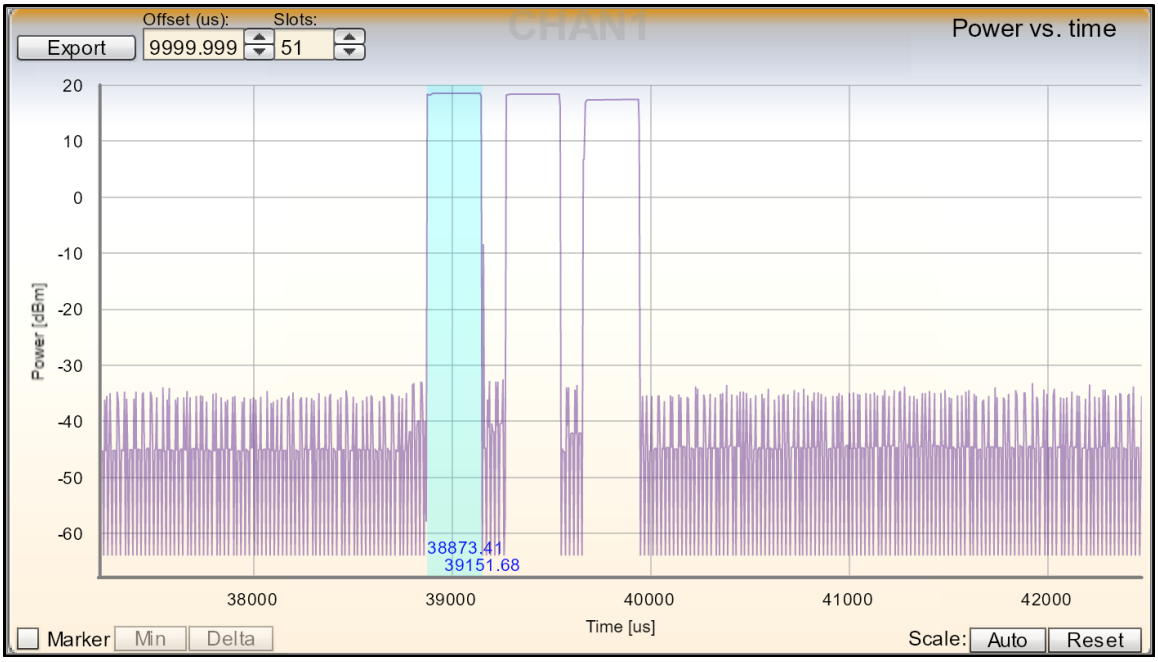
For example, one of the more volatile settings that also happened to be vital to intercepting packet is the frequency setting. As different technologies and protocols utilize different frequency bands, one would think that switching between different technologies and protocols would also change the frequency being monitored, but this is not the case for the IQxel-M16W. Specifically, while the IQxel-M16W can officially switch between modes for different protocols and technologies, the frequency being monitored is determined by three other frequency-related settings. The three frequency settings include: "frequency", "band", and "channel". The "band" settings determine the general frequency spectrum or band being monitored, allowing the IQxel-M16W to switch between the 2.4 GHz band and a variety of U-NII bands. The "frequency" and "channel" settings are essentially two different methods to accomplish the same goal, which is to set the center frequency to be monitored. While the "frequency" option allows the user to input a frequency, the "channel" option lets the user to choose from a list of predetermined channels based on the respective technology/protocol channel map. Thus, in order to capture BR/EDR and BLE packets, the IQxel-M16W must first be switched to Bluetooth mode with the "band" settings set to monitor 2.4 GHz band. Special care must be taken when it comes to adjusting the center frequency using either the "frequency" or "channel" settings as it may caused dropped packets. Particularly, in tests conducted with the Cypress EVB transmitting BLE advertising packets on each of the three advertising channels 37, 38, and 39 at 2402 MHz, 2426 MHz, and 2480 MHz, respectively, it is discovered that setting the center frequency too close to either end of the Bluetooth band could cause packets transmitted in the other end of the frequency band to be dropped. Generally, for all later tests, the center frequency was set to a frequency in the middle of the band, usually either 2426 MHz or 2440 MHz, to avoid packets being dropped.

Similar to the how the frequency is set, the power level is also controlled by more than just one settings. These settings include "reference level", "expected nominal power", and "user margin". It was previously discovered by a colleague and an engineer at octoScope that the signals were being distorted by the Automatic Gain Control (AGC) function. In particular, the signals were being clipped because the AGC function was setting "reference level" to just a little bit over the noise floor at -40 dBm. This is much lower than actual

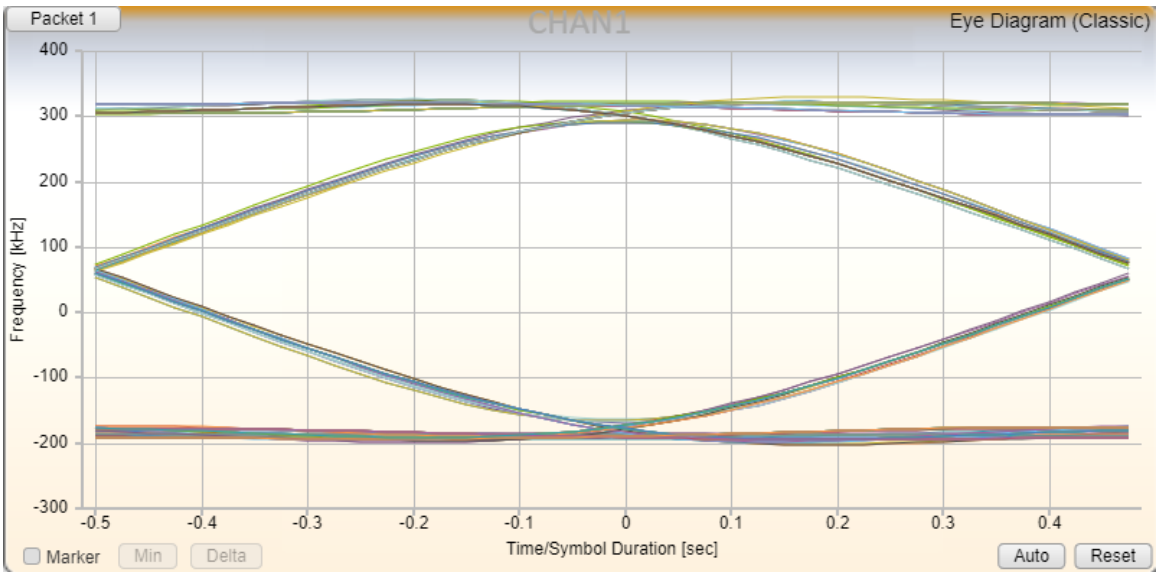
received signal strength of +15dBm with the EVB connected directly into one of the IQxel-M16W's analyzer port instead of using an antenna. It should be noted that in contrast to the frequency settings, there is no gimmick involve in tuning the other two settings. The "expected nominal power" value should always be around -2 dBm lower than the reference level and the value for "user margin" is just the difference between "reference level" and "expected nominal level".

Outside of the frequency settings and the reference power level settings, the only other settings on the IQxel-M16W that really affect packet capture operation are "sampling rate", "capture length", and "data rate". The self-explanatory "sampling rate" setting is always set to double the bandwidth of the Bluetooth frequency spectrum at 160 MHz according to the Nyquist-Shannon sampling theorem. The "capture length" settings determine the length of time or duration of a capture conducted by the IQxel-M16W. Even though the IQxel-M16W provide the user the option to choose between "single capture" and "continuous capture" mode, the "continuous capture" is multiple capture instances with duration as specified by the capture length field. Usually, the IQxel-M16W is set to single capture mode with a "capture length" of 100 ms. Unlike as the two aforementioned settings, the "data rate" setting allow the user to choose between a wide variety of data rate options for both BR/EDR and BLE technology instead of requiring an input value. For this option, the default "auto-detect" option works fine with both BR/EDR and BLE. Alternatively, for BLE captures, the specialized "LEnergy" data rate option also works adequately.

Once all of the aforementioned settings have been set correctly, the IQxel-M16W can start intercepting and capturing packets. With the packets captured, the web-interface built-in graphing and logging function can provide results in the form of eye diagrams, power vs time graphs, adjacent channel power graphs, 20 db bandwidth graphs, as well transmit quality and demodulator output summary. Although the visualizations and logs are helpful in confirming and assuring the quality of the capture, they have several shortcomings that diminish their usefulness in overall system. One of the most troublesome shortcomings is the lack of adaptation and adjustment of the web-interface for Bluetooth technologies. In particular, within the web-interface, only packets that are transmitted on the same frequency as the center frequency set in either the "frequency" or "band" settings is decoded.



(a) Power vs Time Graph



(b) Eye Diagram

Figure 4.2: Result graph and visualization for the EVB direct connect test. Since the EVB was connected directly to the Litepoint engine, the signal strength was quite high (around 20dBm) and the eye diagram is very clear.

The lack of adaptation is especially problematic for BLE as it was shown that in tests conducted with BLE transmission, the IQxel-M16W still recognize the packets captured as those of the classic BR/EDR pattern instead. Furthermore, there is no way to set the center frequency with the "channel" settings using the 40-channel BLE channel map, when only the 79-channel BR/EDR channel map is available for use. In addition to all the aforementioned deficiencies, the web-interface also does not possess the ability to log the captured packets to a more standard and universal platform such as PCAP or PCAPNG.

Thus, while the web-interface is counted on to initiate the packet capture operation, the packet are processed and logged through the use a Python scripts in conjunction with Standard Commands for Programmable Instruments (SCPI) console commands that allow us to access functions unavailable in the web-interface. Separate versions of the aforementioned script exist for classic BR/EDR and BLE, respectively. The BR/EDR script was developed by an enginner at octoScope based on a script orginally used for Wi-Fi packet captures. Later on, the same script used for classic BR/EDR captures were modified in [54] with some assistance from Litepoint to accommodate BLE packet captures. Basically, both the BR/EDR and BLE script is supposed to extract the captured packets from the IQxel-M16W, process them, and log them into a PCAPNG file. It is worth noting that utilizing the scripts along with SCPI commands not only make it possible to log packets into standard packet capture format such as PCAPNG but it also resolve some of the issues encountered with the web-interface. In particular, SCPI commands provided by Litepoint allow the IQxel-M16W to decode and analyse all packets irrespective of the channel they were transmitted. Moreover, the SCPI commands also unlocks the ability to properly recognize BLE packets, allowing these packets to be properly decoded and logged. Figure 4.2a, and b showcase the power/time graph, and the eye diagram from a successful direct connect capture.

4.2 Litepoint IQxel-MW 7G Based Promiscuous Packet Capturing Subsystem

With some promising initial results, it was decided the project is ready to move on to the next phase. One of the primary objectives for the next phase is to modify and reconstruct

the system to capture over-the-air BR/EDR and BLE traffic as all previous test thus far have been conducted with the transmitter circuit directly connected to the SDR. Another important objective is to adapt the system to new hardware in the form of the Litepoint IQxel-MW 7G (see Figure 4.3).

Figure 4.3: The Litepoint IQxel-MW 7G Used in Later Builds of the Promiscuous Packet Capture System [9]. Outwardly, there are no major differences between the IQxel-M16W and the IQxel-MW 7G as the layout are exactly the same.

One of the driving forces behind the hardware changes was the release of the new Wi-Fi standard IEEE 802.11ax, also commonly known as Wi-Fi 6 [68]. Whereas the IQxel-M16W, being an older platform, was ill suited to support the new standard, the newer IQxel-MW 7G model was specifically designed with Wi-Fi 6 testing in mind. As previously mentioned, since the project was a part of larger undertaking to develop to develop a wireless test-bed for technologies utilizing the 2.4 GHz band, the hardware change was necessary to accommodate the larger effort. Furthermore, we also hoped that an updated and modern platform would also resolve some of the issues encountered with the M16W.

In order to capture over-the-air Bluetooth traffic, significant changes to the previous test set-up were necessary. While special care was always taken to ensure that the Litepoint engine was properly shielded and isolated, serious consideration was never given to the

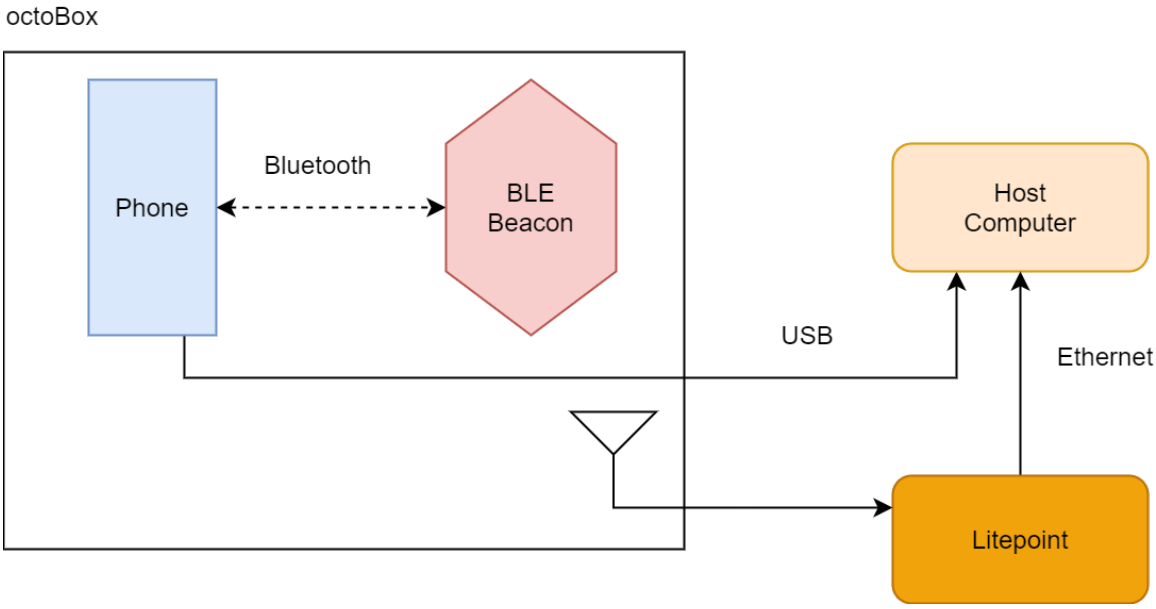
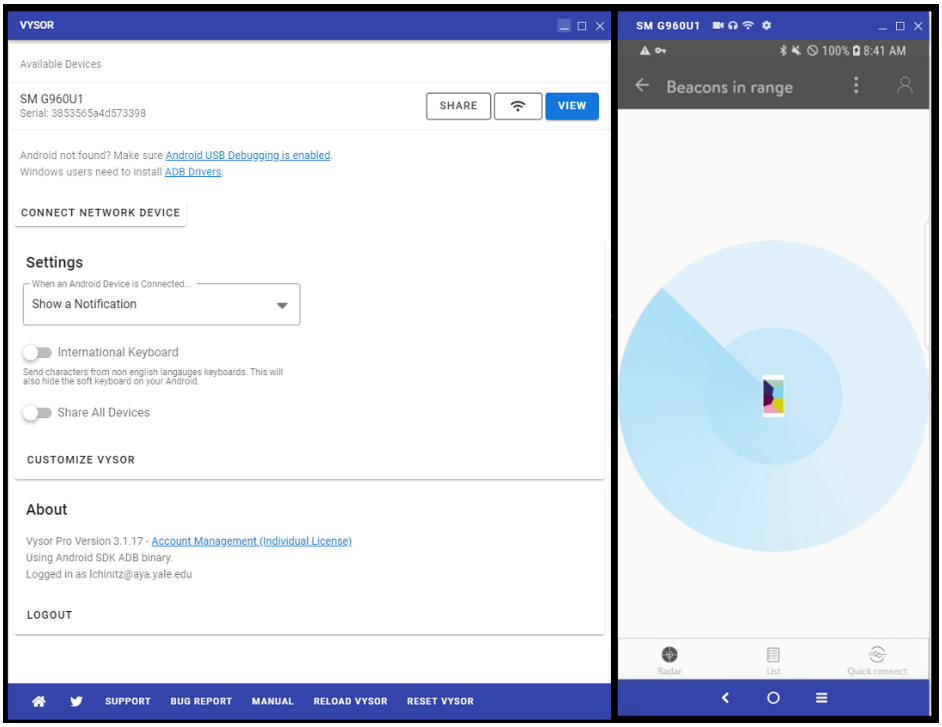


Figure 4.4: The Over-the-air advertising beacon test setup for the Litepoint IQxel-7G-Based Promiscuous Packet Capture System. Shielding is now necessary as the Bluetooth packets are transmitted over-the-air. Furthermore, an actual BLE beacon now replace the EVB in the test scenario.

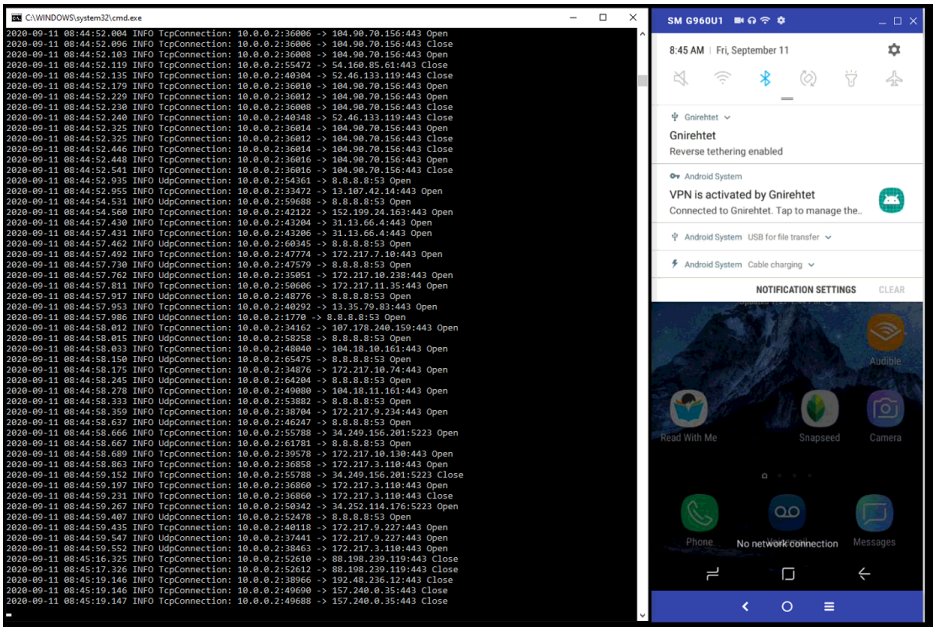
shielding and isolation for the transmitter device as all previous tests were conducted with either the transmitter connected directly to the SDR or the SDR looping back to itself. As one of the primary detriment to over-the-air transmission is interference from other devices utilizing the same frequency, shielding and isolation for the transmitter device was deemed necessary to assure the quality of the capture. Thus, all transmitter devices used in the tests are placed within a semi-anechoic test chamber designed by octoScope (see Figure 4.4). With the test chamber in the equation, a host of new issues needs to be addressed. For instance, interacting with test devices that require manual control and input suddenly such as smartphones and tablet became impossible as these devices need to be isolated for the entire duration of the capture session. Furthermore, utilizing devices that require an active network connection is also an issue as the whole point of the isolation chamber is to keep other RF signals out. The solution to the interaction issue was relatively simple as there are software programs that allow the user to control the targeted Android or iOS device from the host computer through a USB connection. However, the network issue

was a much tougher problem to address as there was simply no straightforward method to provide a network connection within a isolation chamber. The first proposed solution was to use a Ethernet-to-USB-C adapter to provide the test device a connection through Ethernet. Despite early promising results, it was soon discovered that even the most efficient and well-built adapters were plagued by battery and heat issue. To be more specific, the Ethernet adapters were draining power at a rate that the charger simply could not keep up. Furthermore, it was discovered that while the connection was active, the adapters were also dissipating an unacceptably high amount of heat which pose a fire hazard in the enclosed confine of the isolation chamber. Consequently, an alternate solution was found in the form of the technique called "reverse-tethering" [69]. In contrast to the concept of tethering in which the smartphone's network connection is shared with a connected computer, reverse-tethering instead seeks to share the connected computer's network connection with the smartphone (see Figure 4.5). Although Android previously possessed a built-in "reverse-tethering" function, this was removed in later versions due to a lack of interest and thus an open-sourced reverse-tethering tool was used instead. Since a reverse tethered connection does not require the phone to power the adapter on top of its functionality, there is neither a battery nor a heat issue.

With the test-setup in order, the only task left was to adjust and fine-tune the settings in the web-interface to initiate a capture. Since no significant changes were made to the web-interface, parameters and values previously utilized to configure the IQxel-M16W are also for the most part migrated to the IQxel-MW 7G . Similar to how the IQxel-M16W was initiated, first the IQxel-MW 7G must first be switched to Bluetooth mode. For the frequency settings, the "band" settings must be set to monitor the 2.4 GHz band while the center frequency is set to a frequency in the middle of the Bluetooth spectrum, normally 2426 MHz or 2440 MHz, using either the "frequency" or the "channel" settings. As for the reference power level settings, AGC must first be disabled before other power level settings are manually adjusted. Since the average received signal strength of BLE transmissions in our tests is around -17 dBm, the "expected nominal power" value is set accordingly to the same level while the "reference level" value is set to be about 2 dBm lower. The "sampling rate" and "capture length" options are once again set to 160 MHz and single

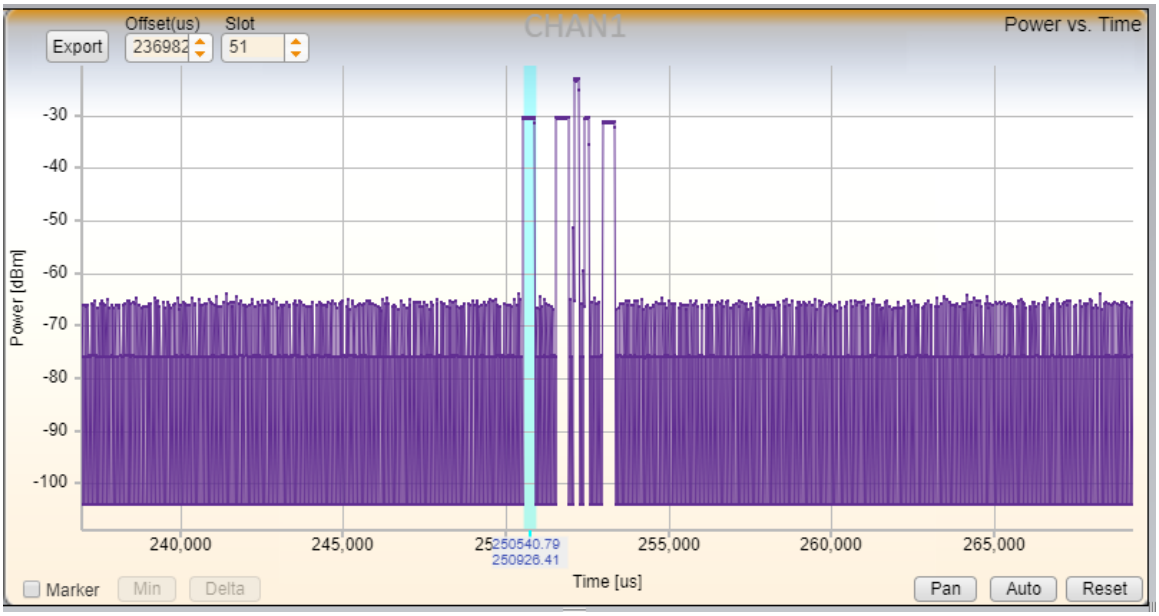


(a) Vysor Software For USB-based Remote Control of Android Phones

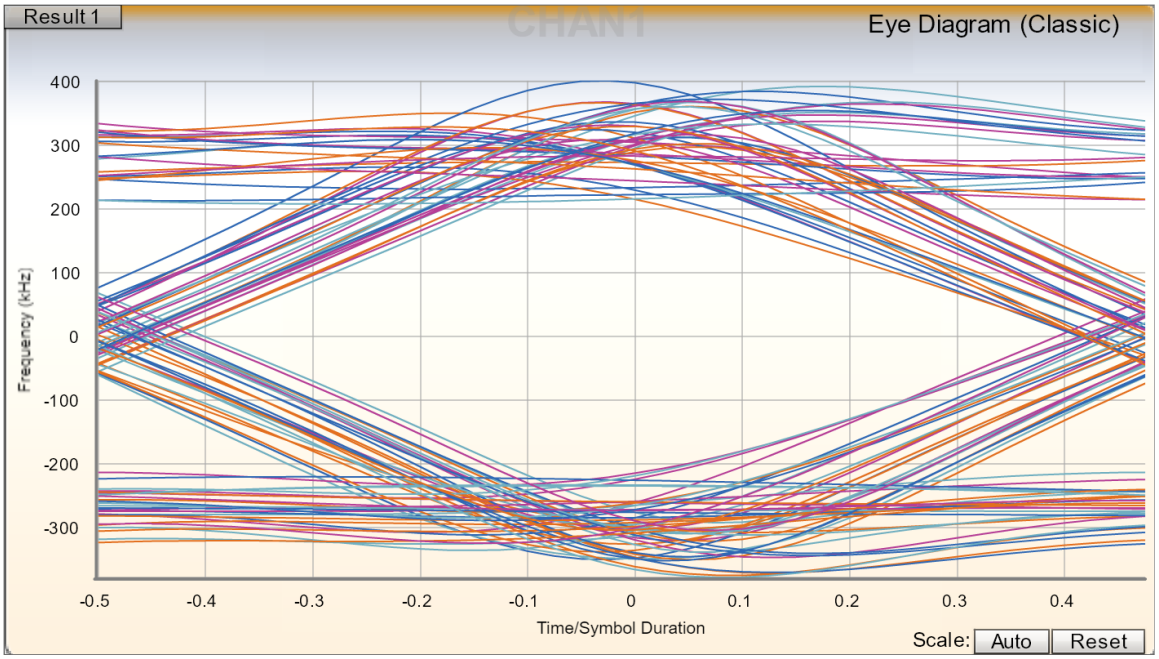


(b) Reverse Tethering Software for USB-based Ethernet for Android Phones

Figure 4.5: Picture of the UI of softwares used for remote operation and providing network connection to RF-isolated smartdevices



(a) Power vs Time Graph



(b) Eye Diagram

Figure 4.6: Result Graph and Visualization for the Over-the-air Advertising Beacon Test. As the Beacon is not directly connected to the Litepoint, the signal level is significantly lower (around -30 dBm) and the shape of the eye diagram ,while still resembling the shape of the eye, is also more erratic.

capture/single capture, respectively, mirroring the settings used to configure IQxel-M16W. As the “LEnergy” option was not working for some reason, the “data rate” setting was set to the default “auto-detect” option for all tests conducted. It is worth noting that, aside from the settings mentioned above, the “ATP Type” settings, which specify the number of channels to be calculated in adjacent channel power measurement, must also be manually adjusted before the IQxel-MW 7G can start capturing Bluetooth packets. Specifically, there are three main option for the “ACP type” settings: BW10M, BW20M, and ACH, which uses a +/- 5 MHz bandwidth from the carrier, +/-10 MHz bandwidth from the carrier, and all 79 BR/EDR channels to calculate the adjacent channel power measurement. Whereas all three options would allow the IQxel-M16W to analyse BLE packets, only the BW10M permits the same thing on the IQxel-MW 7G. With all the settings has been correctly set and the capture finished, the captured packets are passed through the same logger script used for the IQxel-M16W to be processed and logged. Figure 4.6a, and b showcase thepower/time graph, and the eye diagram from a successful direct connect capture.

4.3 Proposed Hybrid System

While the previous tests indeed proved that it is possible to capture, decode, and log Bluetooth packets with the Litepoint product test engine, this was only true without taking encryption and security mechanism into consideration. To be more exact, while the system on its own would have had no issue with unencrypted advertising packets that are transmitted using the same three channels every time,the same cannot be said for fully encrypted data packets that are transmitted on a randomized sequence of frequencies. Originally, it was assumed that there would be a method or function within the Cypress API that would allow the user to extract the encryption key and frequency jump pattern from the IC. However, this direction was proven flawed, as upon inquiry, Cypress disclosed that no such function currently exist as of yet and there would also be no plans to include such a function in future updates. Thus, we were forced to turn to an alternate solution in which another promiscuous sniffer would work in tandem with the Litepoint engine. Specifically, a promiscuous subsystem built around a Nordics Semiconductor IC [10], one that possess

the ability to extract the encryption key and the jump pattern, would help the Litepoint handle the data packets (see Figure 4.7).

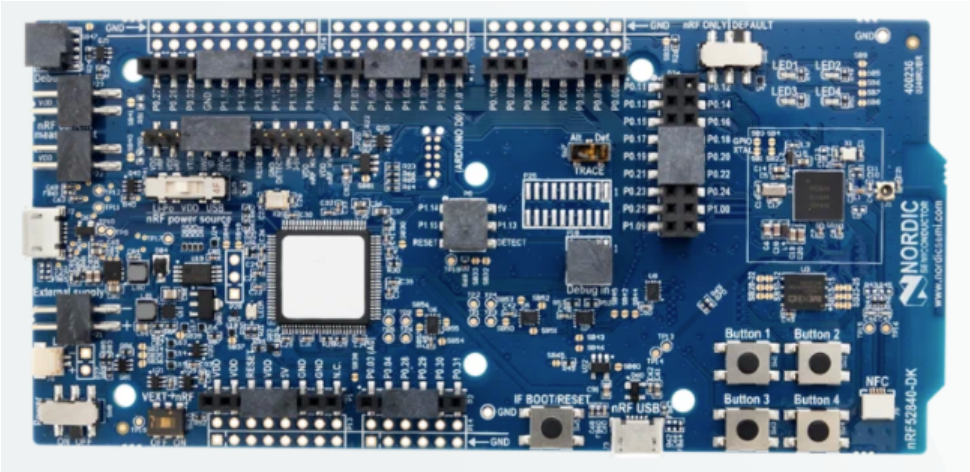


Figure 4.7: The Nordics Semiconductor nRF52840 DK EVB used in the Complementary Promiscuous Packet Capture System. The nRF52840 can be connected to a host computer via a USB-B connection and it also possess built-in antenna on the right-side of the board. [10]

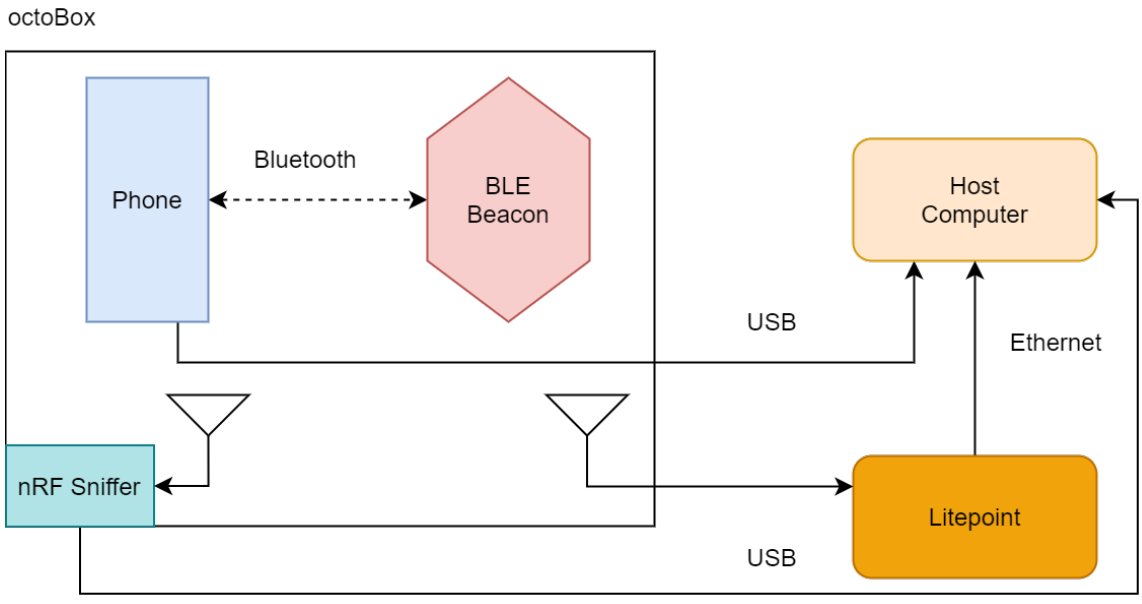


Figure 4.8: The Over-the-air advertising beacon test setup for the Litepoint IQxel-7G-Based Promiscuous Packet Capture System with the nRF52840-based Complementary Promiscuous Packet Capture Subsystem

The results for this direction seems promising as the extraction of the credentials needed to process data packets was relatively straightforward and simple. Furthermore, since the Nordics sniffer tool log all of the intercepted packets to the standard PCAPNG format by default, this subsystem also provide another useful reference point that can be used to crosscheck packet capture file logged by both the inline Cypress IC based system and the promiscuous Litepoint based system. Figure 4.8 display the test setup detailed in this part.

At the time of the writing, this supplementary promiscuous sniffer subsystem remains mostly a concept as adequate testing has not been conducted and many functions in their current build require manual input and control. Future work will seek to test whether the credentials extracted can help the Litepoint subsystem decrypt the packet and predict the jump pattern as well as to streamline and automate the process further. Figure 4.9 showcase a successful capture using the nRF52480.

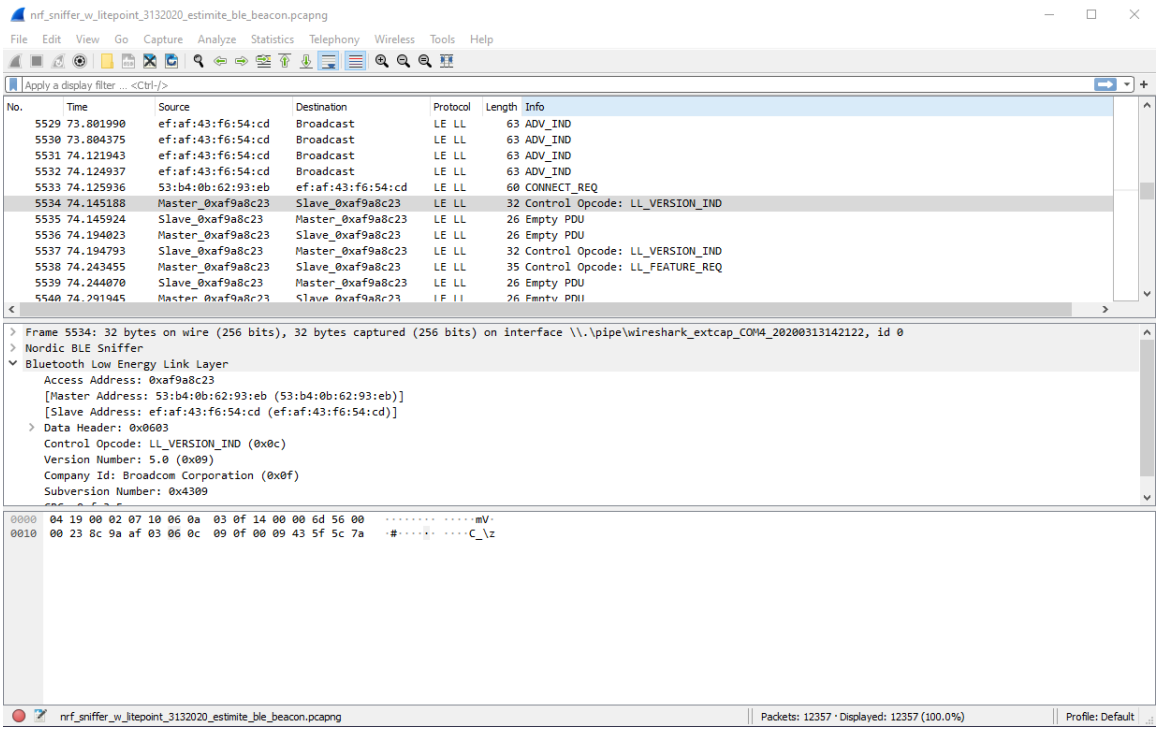


Figure 4.9: Example of a successful capture logged using the nRF52480 DK EVB. This section of the capture specifically detail the moment the handshake Between the Android phone and the beacon completed. Note how the MAC address of the phone and the beacon in the Source field is changed to Master and Slave after the handshake completed.

4.4 Summary

This chapter explored the development process of the Litepoint-based promiscuous sniffer subsystem. Specifically, each section is dedicated to detailing the technical data for the two versions of the subsystem as well as design choices made and challenges faced during the development process. Whereas the Section 4.1 mainly focused on a previous build of the system and how to replicate past results, Section 4.2 and 4.3 discussed what the present system looked like and how a refined version of the system would flesh out, respectively. The project has managed to demonstrate the viability of a promiscuous sniffer based on the Litepoint production test engine as well as the concept of a hybrid inline/promiscuous system that not only monitor over-the-air traffic but also provide comprehensive information about the RF environment in the 2.4 GHz spectrum. Although the ability to intercept and log data packets is still in the development and early testing phase at the time of writing, initial tests show promise and the ultimate goal of the project is well within reach.

Chapter 5

Conclusion

Overall, the project has successfully achieved its intended objectives. The packet capture file formatting tool has demonstrated the ability to log packets to not only the standard BTSNoop format but also the universal PCAP, and PCAPNG format as well. Furthermore, the Litepoint-based promiscuous sniffer has been shown to be able to reliably capture and log unencrypted over-the-air BR/EDR, and LE traffic. Additionally, the complementary subsystem that handle decryption has also reached initial operational capability and would soon be ready with further testing, and refinement. Technical details and challenges encountered during the development process of the logging tool and the promiscuous sniffer are detailed in the chapters of this thesis. Chapter 2 presented an overview of background technical information on the various classes of Bluetooth-derived protocols with special emphasis on BR/EDR and BLE. Moreover, specialized software, hardware, and the standard file formats utilized in packet sniffing tools were also explored in Chapter 2. The technical details and the development process of the packet formatting and logging function are showcased in Chapter 3. Additionally, relevant specifications and data of the inline packet capture system that the logging function was tested upon were also provided. Finally, Chapter 4 describe both the past and present builds of the Litepoint-based promiscuous packet capture system as well as an complementary subsystem under development that is supposed to help the Litepoint deal with encryption and frequency-hopping patern.

5.1 Research Outcomes

The work presented in this thesis has resulted in the following research outcomes:

- Software that process and logs captured BR/EDR and BLE packets into standard and universal packet capture file format such as PCAP and PCAPNG were successfully developed and demonstrated.
- The viability of the Litepoint-based promiscuous packet capture system was proven as the current build of the system has successfully intercepted and logged BR/EDR and BLE packets that was transmitted over-the-air.
- A complementary subsystem that would assist the Litepoint-based system in dealing with encrypted packets that randomly hop between frequencies. Initial tests shows promise but further development and refinement needed before assimilation.

5.2 Future Work

Given the outcomes of this work, the following future tasks are of interest to the authors:

- Address the shortcomings and issues for the logging system and the promiscuous sniffer system in outlined in Chapter 3 and Chapter 4.
- Complete development of the complementary system that extract encryption key and frequency hopping pattern.
- Integration of the hybrid promiscuous/inline Bluetooth packet capture system with its Wifi counterpart as well as those pertaining to other wireless communication standard.

Bibliography

[1] Bluetooth SIG, "Bluetooth market update 2018," Bluetooth SIG, 2018. [Online]. Available: https://www.bluetooth.com/bluetooth-resources/2018-bluetooth-market-update/

[2] ——, "Bluetooth market update 2019," Bluetooth SIG, 2019. [Online]. Available: https://www.bluetooth.com/bluetooth-resources/2019-bluetooth-market-update/

[3] ——, "Bluetooth market update 2020," Bluetooth SIG, 2020. [Online]. Available: https://www.bluetooth.com/bluetooth-resources/2020-bmu/

[4] octoScope, "lstack-benchtop," octoScope, 2012. [Online]. Available: https://www.octoscope.com/English/Products/Ordering/Testbeds_PreConfigured/STACK-BENCHTOP.html

[5] Lambda Laboratory Instruments , "Rs-232 connection cable," Lambda Laboratory Instruments, Flickr, 2012. [Online]. Available: https://www.flickr.com/photos/50299145@N03/7189464433

[6] U. Wetzker, I. Splitt, M. Zimmerling, C. A. Boano, and K. Römer, "Troubleshooting wireless coexistence problems in the industrial internet of things," in *2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES)*, 2016, pp. 98–98.

[7] Cypress Semiconductor, "Cyw920719q40evb-01 evaluation kit," Cypress Semiconductor, 2020. [Online]. Available: https://www.cypress.com/documentation/development-kitsboards/cyw920719q40evb-01-evaluation-kit

[8] Litepoint, "Iqxel-mw," Litepoint, 2020. [Online]. Available: https://www.litepoint.com/products/iqxel-mw/

[9] ——, "Iqxel-mw," Litepoint, 2020. [Online]. Available: https://www.litepoint.com/products/iqxel-mw-7g/

[10] Nordics Semiconductor, "nrf52840 dk," Nordics Semiconductor, 2019. [Online]. Available: https://www.nordicsemi.com/Software-and-tools/Development-Kits/nRF52840-DK

[11] Product Focus, "Top 25 inventions of the last 25 years," Product Focus, 2019. [Online]. Available: https://www.productfocus.com/top-25-inventions-of-the-last-25-years/

[12] Chris Anderson, "The wi-fi revolution," Wired, 2003. [Online]. Available: https://www.wired.com/2003/05/wifirevolution/

[13] Sai Krishna, "Importance of wireless technology," PC Tablet, 2017. [Online]. Available: https://pc-tablet.com/importance-wireless-technology/

[14] Roy Blunt, "The importance of 5g," Senate Republican Policy Committee, 2019. [Online]. Available: https://www.rpc.senate.gov/policy-papers/the-importance-of-5g

[15] IEEE, "The importance of 5g," IEEE Transmitter, 2020. [Online]. Available: https://transmitter.ieee.org/why-the-development-of-wireless-networks-is-important-for-global-iot-growth/

[16] Bluetooth SIG, "Specifications of bluetooth systems," Bluetooth SIG, 2019.

[17] Apple, *AirPods (1st generation) - Technical Specifications*, Apple, 2019. [Online]. Available: https://support.apple.com/kb/SP750?locale=en_US

[18] Google, *Contanct Tracing-Bluetooth Specifications*, Google, 2020. [Online]. Available: https://blog.google/documents/58/Contact_Tracing_-_Bluetooth_Specification_v1.1_RYGZbKW.pdf

[19] E. Ferro and F. Potorti, "Bluetooth and wi-fi wireless protocols: a survey and a comparison," *IEEE Wireless Communications*, vol. 12, no. 1, pp. 12–26, 2005.

[20] Mikołaj Skawiński, "Bluetooth vs. wi-fi," Netguru, 2019. [Online]. Available: https://www.netguru.com/codestories/bluetooth-vs-wifi-comparison-for-the-iot-solutions

[21] John Lukez, "Test challenges of wi-fi and bluetooth devices," Litepoint and Evaluation Engineering, 2020. [Online]. Available: https://www.evaluationengineering.com/home/article/13004345/test-challenges-of-wifi-and-bluetooth-devices

[22] Laura Wood, "Global $14.5bn wireless testing market by offering, technology, application and region - forecast to 2024," Research and Market, 2019. [Online]. Available: https://www.globenewswire.com/news-release/2019/10/21/1932383/0/en/Global-14-5Bn-Wireless-Testing-Market-by-Offering-Technology-Application-and-Region-Forecast-to html

[23] Mark Loveless, "Bluetooth hacking tools comparison," Duo, 2017. [Online]. Available: https://duo.com/decipher/bluetooth-hacking-tools-comparison

[24] octoScope, "Triathlon - rf/mac/phy analyzer," octoScope, 2019. [Online]. Available: https://www.octoScope.com/English/Products/Ordering/Testbed_Building_Blocks/Triathlon.html

[25] National Instrument, "Introduction to bluetooth device testing," National Instrument, 2016. [Online]. Available: http://download.ni.com/evaluation/rf/intro_to_bluetooth_test.pdf

[26] Mohammad Afaneh, "How to use a bluetooth sniffer without pulling your hair out!" NovelBits, 2016. [Online]. Available: https://www.novelbits.io/bluetooth-low-energy-sniffer-tutorial/

[27] Michael Ossmann, "Project ubertooth: Building a better bluetooth adapter," Project Ubertooth, 2011. [Online]. Available: http://ubertooth.sourceforge.net/hardware/one/

[28] Texas Instrument, "Cc2540emk-usb," Texas Instrument, 2010. [Online]. Available: https://www.ti.com/tool/CC2540EMK-USB

[29] Nordics Semiconductor, "nrf52840 dongle," Nordics Semiconductor, 2018. [Online]. Available: https://www.nordicsemi.com/Software-and-tools/Development-Kits/nRF52840-Dongle

[30] Cypress Semiconductor, "Cysmart – bluetooth® le test and debug tool," Cypress Semiconductor, 2018. [Online]. Available: https://www.cypress.com/documentation/software-and-drivers/cysmart-bluetooth-le-test-and-debug-tool

[31] Teledyne-Lecroy, "Frontline bpa® low energy bluetooth® protocol analyzer," Teledyne-Lecroy, 2016. [Online]. Available: https://www.nordicsemi.com/Software-and-tools/Development-Kits/nRF52840-Dongle

[32] Travis F. Collins, Robin Getz, Di Pu, Alexander M. Wyglinski, *Software-Defined Radio for Engineers*. Artech House, 2018.

[33] Bluetooth SIG, "Validated and recognized test equipment," Bluetooth SIG, 2020. [Online]. Available: https://www.bluetooth.com/develop-with-bluetooth/qualification-listing/qualification-test-tools/validated-recognized-test-equipment/

[34] Teledyne-Lecroy, "Sodera series of bluetooth protocol analyzers," Teledyne-Lecroy, 2017. [Online]. Available: https://www.fte.com/products/sodera.aspx/

[35] Ellisys, "Ellisys bluetooth explorer all-in-one bluetooth® analysis system," Ellisys, 2018. [Online]. Available: https://www.ellisys.com/products/bex400/

[36] ——, "Ellisys bluetooth tracker ultra-portable ble and wi-fi protocol analyzer," Ellisys, 2018. [Online]. Available: https://www.ellisys.com/products/bex400/

[37] Texas Instrument, "Packet sniffer," Bluetooth SIG, 2014. [Online]. Available: https://www.ti.com/tool/PACKET-SNIFFER

[38] Google, "Verifying and debugging bluetooth," Google, 2020. [Online]. Available: https://source.android.com/devices/bluetooth/verifying_debugging

[39] Teledyne Lecroy, "Btsnoop file format," Teledyne Lecroy, 2010. [Online]. Available: http://www.fte.com/webhelp/bpa600/Content/Technical_Information/BT_Snoop_File_Format.htm

[40] Wireshark, *Libpcap File Format*, Wireshark, 2015.

[41] M.Tuexen, F.Risso, J.Bongertz, G.Combs, G.Harris, M.Richardson, *PCAP Next Generation Capture File Format*, IETF, 2020.

[42] octoScope, "Small anechoic chamber channels: Estimating channel capacity from a chamber model," octoScope, 2019. [Online]. Available: https://www.octoScope.com/English/Resources/Whitepapers.html

[43] ——, "Testbed overview," octoScope, 2020. [Online]. Available: https://www.octoScope.com/English/Collaterals/Documents/octoBox_personal_testbed.pdf

[44] ——, "Products offered," octoScope, 2020. [Online]. Available: https://www.octoscope.com/English/Products/Ordering/index.html

[45] M. T. Dini, V. Sokolov, and V. Buriachok, "Men-in-the-middle attack simulation on low energy wireless devices using software define radio," *CoRR*, vol. abs/1906.10878, 2019. [Online]. Available: http://arxiv.org/abs/1906.10878

[46] Q. Yang and L. Huang, *Bluetooth Security*. Singapore: Springer Singapore, 2018, pp. 195–226. [Online]. Available: https://doi.org/10.1007/978-981-10-8447-8_6

[47] F. Mazzenga, D. Cassioli, P. Loreti, and F. Vatalaro, "Evaluation of packet loss probability in bluetooth networks," in *2002 IEEE International Conference on Communications. Conference Proceedings. ICC 2002 (Cat. No.02CH37333)*, vol. 1, 2002, pp. 313–317 vol.1.

[48] W. Albazrqaoe, J. Huang, and G. Xing, "A practical bluetooth traffic sniffing system: Design, implementation, and countermeasure," *IEEE/ACM Trans. Netw.*, vol. 27, no. 1, p. 71–84, Feb. 2019. [Online]. Available: https://doi-org.ezpxy-web-p-u01.wpi.edu/10.1109/TNET.2018.2880970

[49] Bluetooth SIG, "Test equipment," Bluetooth SIG, 2020. [Online]. Available: https://www.bluetooth.com/develop-with-bluetooth/build/service-providers/test-equipment/

[50] Cory Doctorow, "Anechoic chamber 1," Consumers Union Labs, Flickr, 2015. [Online]. Available: https://www.flickr.com/photos/doctorow/22337544326

[51] Movax, "High-voltage lab control room," Kettering University, Flickr, 2010. [Online]. Available: https://www.flickr.com/photos/movszx/4756294451/l

[52] Mike Ryan, *Crack LE Bruteforce Decryption Tool*, Crack LE, 2018. [Online]. Available: https://github.com/mikeryan/crackle

[53] Wireshark, "Supported file format for bluetooth," Wireshark, 2015. [Online]. Available: https://wiki.wireshark.org/Bluetooth

[54] G. Wernsing, "Programmable testbed for bluetooth experimentation," Master's thesis, Worcester Polytechnic Institute, 100 Institute Rd, Worcester MA, 12 2019.

[55] John Herman, "Why everything wireless is 2.4 ghz," Wired, 2010. [Online]. Available: https://www.wired.com/2010/09/wireless-explainer/

[56] Kaspersky Lab, "What is a packet sniffer?" Kaspersky Lab, 2017. [Online]. Available: https://www.lifewire.com/definition-of-sniffer-817996

[57] Bradley Mitchell, "What is a network sniffer?" Lifewire, 2019. [Online]. Available: https://www.lifewire.com/definition-of-sniffer-817996

[58] Kevin J. Connolly, *Law of Internet Security and Privacy.* Aspen Publisher, 2003.

[59] Frank Golatowski, "Bluetooth baseband," Universität Rostock - Institut für Angewandte Mikroelektronik und Datentechnik, 2001. [Online]. Available: https://www.amd.e-technik.uni-rostock.de/ma/gol/lectures/wirlec/bluetooth_info/baseband.html

[60] Bluetooth SIG, "Learn about bluetooth-radio versions," Bluetooth SIG, 2017. [Online]. Available: https://www.bluetooth.com/learn-about-bluetooth/bluetooth-technology/radio-versions/

[61] Basavaraj Patil, Yousuf Saifullah, Stefano Faccin, Srinivas Sreemanthula, Lachu Ar-avamudhan, Sarvesh Sharma, Risto Mononen, *IP in Wireless Networks*. Prentice Hall, 2003.

[62] Texas Instrument, "Bluetooth low energy protocol stack basics," Texas Instrument, 2017. [Online]. Available: http://software-dl.ti.com/lprf/simplelink_cc2640r2_latest/docs/blestack/ble_user_guide/html/ble-stack-3.x/overview.html

[63] MathWorks, "Bluetooth protocol stack," MathWorks, 2019. [Online]. Available: https://it.mathworks.com/help/comm/ug/bluetooth-protocol-stack.html

[64] Microchip Technologies, "Bluetooth low energy packet types," Microchip Technologies, 2016. [Online]. Available: https://microchipdeveloper.com/wireless:ble-link-layer-packet-types

[65] RF Wireless World, "Ble advertising and ble data packet format," RF Wireless World, 2019. [Online]. Available: https://www.rfwireless-world.com/Terminology/BLE-Advertising-and-Data-Packet-Format.html

[66] Wireshark, "Wireshark frequently asked questions," Wireshark, 2020. [Online]. Available: https://www.wireshark.org/faq.html#q1.2

[67] Omar Santos, Panois Kampanakis, Aaron Woland, "Introduction to and design of cisco asa with firepower services," Cisco, 2016. [Online]. Available: https://www.ciscopress.com/articles/article.asp?p=2730336&seqNum=2

[68] Wi-Fi Alliance, "Wi-fi certified 6," Wi-Fi Alliance, 2020. [Online]. Available: https://www.wi-fi.org/discover-wi-fi/wi-fi-certified-6

[69] Romain Vimont, "Introducing "gnirehtet", a reverse tethering tool for android," Geny Mobile, 2017. [Online]. Available: https://medium.com/genymobile/gnirehtet-reverse-tethering-android-2afacdbdaec7

Appendix A

Complementary Logger Utility for CYW20719-based Inline Packet Capture Tool

A.1 Control Module

```
import time
#python -m cProfile -s tottime pycontrol.py
import logger_class
import interpreter_class
import communicator_class
import plotter

comms = communicator_class.Communicator()
interp = interpreter_class.Interpreter()
logs = logger_class.Logger()


hci_packets = []
```

```
#reset board
#send queued packets
#go into main loop

###The reset is still not working the way I expect
#returns the body of the first HCI packet
#packet_one = comms.reset_board()
#need to handle the first one manually, since interp doesn't see it
#hci_packets.append([ord(packet_one[0]),packet_one[1:]])

comms.send_initial_packets()


def transfer_a_file():
    global time #I dunno why this needs to be here but not the others
    #SLEEPING WHILE PROCESSING CAN CAUSE THE INPUT BUFFER TO OVERFLOW
    #Also cache everything, since the hard drive can get angry
    data_packets_timing = []
    data_packets = []
    file_name = ""
    while True:
        interp.new_packet()

        header = comms.get_header_bytes()


        len_needed = interp.parse_header(header)
        #check len needed for errors
        #flush buffer to log file in the future
        #change this so that interp handles it silently
        if len_needed < 0:
            comms.clear_input()
            continue

        body = comms.get_packet_bytes(len_needed)
        interp.parse_params(body)
        #print interp
```

```
    if interp.is_HCI_packet():
        pass
        hci_packets.append([interp.get_HCI_direction(),
            interp.get_HCI_packet_bytes()])

    elif interp.file_transfer_requested():
        time1 = time.time()
        print "file transfer started"
        file_name = body[13:-1].rstrip("image/jpe").replace('\n','')
        print "file name: %s" % file_name
        comms.send_packet(interp.get_transfer_req_response())
        file_transfer_started = True
        data_packets_timing.append([time.time(), 0])

    elif interp.is_data_packet():
        data_packets.append(body)
        data_packets_timing.append([time.time(), len_needed])

    elif interp.file_transfer_complete():
        time2 = time.time()
        print "file transfer complete"
        e_time = time2 - time1
        print "elapsed time =", e_time
        break


logs.open_file_transfer(file_name)
for packet in data_packets:
    logs.write_to_file_transfer(packet)
logs.close_file_transfer()

times = [t for t,size in data_packets_timing]
time_zero = times[0]
time_offsets = [x - time_zero for x in times]
sizes = [size for t,size in data_packets_timing]
plotter.show_throughput_plot(time_offsets, sizes, file_name, logs.timestamp)
```

```
num_transfers = 1
#while True:
print "%s transfers expected" % num_transfers
for i in range(num_transfers):
    print "ready for transfer"
    transfer_a_file()
    #waiting at the prompt is blocking, don't leave it waiting here
    #yes_no = raw_input("transfer another file? ")
    #if yes_no[0] not in "yY": break

#HCI packets recieved after completing the final image transfer ARE NOT LOGGED
    CURRENTLY
logs.open_hci_log()
for dir, packet in hci_packets:
    logs.log_HCI_packet(dir, packet)
logs.close_hci_log()
```

A.2 Communicator Module

```
import serial
import serial.tools.list_ports as lp
import time

'''
trace_enable = bytearray.fromhex('19 02 00 02 00 01 01')
set_discoverable = bytearray.fromhex('19 08 00 02 00 01 01')
set_pairable = bytearray.fromhex('19 09 00 01 00 01 ')
get_version_num = bytearray.fromhex('19 02 FF 00 00')
MC_1 = bytearray.fromhex('19 05 23 02 00 13 00')
MC_2 = bytearray.fromhex('19 04 23 02 00 00 02')

commands = [trace_enable, set_discoverable, set_pairable, get_version_num, MC_1,
     MC_2]
```

```
for c in commands:
    print "sending", c
    HCI_UART.write(c)
    stream_input(1)
'''

trace_enable = '19 02 00 02 00 01 01'
set_discoverable = '19 08 00 02 00 01 01'
set_pairable = '19 09 00 01 00 01 '
get_version_num = '19 02 FF 00 00'
set_psm_number = '19 05 23 02 00 13 00'
set_mcu_size = '19 04 23 02 00 00 02'

commands = [trace_enable, set_discoverable, set_pairable, get_version_num]

class Communicator():

    def __init__(self):

        self.out_packet_queue = [trace_enable, set_discoverable, get_version_num
            , set_pairable]

        correct_port = ""
        #the above depends on your computer etc
        #you need the HCI uart, not the PUART

        #find the correct port based on name
        ports = lp.comports()
        for port in ports:
            if port[1].startswith("WICED HCI UART"):
                correct_port = port[0]
                break
        else:
            print "no WICED HCI port found"
            #this will need to change eventually
            quit()
```

```
        print "found WICED HCI:", correct_port


        self.HCI_UART = serial.Serial(correct_port, 3000000, timeout=1)
        #1s timeout, assumes a full packet will be transmitted within 1 s
        time.sleep(.100)
        #pyserial doens't hold until the port is actually ready
        #meaning it doesn't flush properly if you don't wait

        self.flush()



    #returns the header, if there is one
    def get_header_bytes(self):
        if self.HCI_UART.in_waiting >= 5:
            return [ord(x) for x in self.HCI_UART.read(5)]
        else:
            return False


    def get_packet_bytes(self, reqd_length):
        return [ord(x) for x in self.HCI_UART.read(reqd_length)]


    def send_packet(self, packet_bytes_string):
        self.HCI_UART.write(bytearray.fromhex(packet_bytes_string))

    def send_queued_packet(self):
        message = self.out_packet_queue.pop(0)
        self.send_packet(message)
        return message

    def add_to_queue(self, packet_bytes_string):
        self.out_packet_queue.append(packet_bytes_string)
```

```
    def num_out_queued(self):
        return len(self.out_packet_queue)

    def flush(self):
        if self.HCI_UART.in_waiting > 0:
            self.HCI_UART.reset_input_buffer()
            print "cleared input"
            #if this starts in the middle of a RX, things may go sideways

    def read_all_input_buffer(self):
        return [ord(x) for x in self.HCI_UART.read(self.HCI_UART.in_waiting)]

    def close_port(self):
        self.HCI_UART.close()
```

A.3 Interpreter Module

```
'''
format:
group found by dictionary lookup
event found by array index in group
each event is an array of parameter lengths
variable lengths are left off, as they always are at the end
and the params interpreter just appends any unused length

this version has no support for translating to english
it also doesn't play nice with HCI format messages

'''

no_command = [] #no command for code 0x00

#groups

################################################################################
```

```
command_status     = [1]
WICED_trace        = []
HCI_trace          = [1]
NVRAM_data         = [2]
device_started     = []
inquiry_result     = [6,3,1]
inquiry_complete   = []
pairing_completed  = [1,6]
encryption_changed = [1,6]
connected_dev_name = []
user_confirm_req   = [6,4]
device_error       = [1,1]
local_device_addr  = [6]
max_paired         = []
buffer_pool_usage  = [1,2,2,2,2]


DEVICE = [no_command, command_status, WICED_trace, HCI_trace, NVRAM_data,
          device_started, inquiry_result, inquiry_complete,
          pairing_completed, encryption_changed, connected_dev_name,
          user_confirm_req, device_error, local_device_addr,
          max_paired, buffer_pool_usage]

################################################################################
LE = []

################################################################################
GATT = []

################################################################################
HF = []

################################################################################
SPP = []

################################################################################
AUDIO = []
```

```
################################################################################
HIDD = []

################################################################################
AVRC_TARGET = []

################################################################################
TEST = []

################################################################################
TIME = []

################################################################################
ANCS = []

################################################################################
ALERT = []

################################################################################
LN = []

################################################################################
IAP2 = []

################################################################################
AG = []

################################################################################
AIO_CLIENT_SERVER = []

################################################################################
AVRC_CONTROLL = []

################################################################################
AMS = []
```

```
################################################################################

ping_req_reply = []
version_info = [1,1,1,2,3,1]

MISC = [no_command, ping_req_reply, version_info]

################################################################################

connected = []
progress = []
object = []
close = []
access = []
push_data = []

OPS = [connected, progress, object, close, access, push_data]

################################################################################
UNKNOWN =[[] for x in range(256)]
#Codes I don't know

################################################################################

rx_groups = {0x00:DEVICE, 0x01:LE, 0x02:GATT, 0x03:HF,
             0x04:SPP, 0x05:AUDIO, 0x06:HIDD, 0x07:AVRC_TARGET,
             0x08:TEST, 0x0A:TIME, 0x0B:ANCS, 0x0C:ALERT,
             0x0D:LN, 0x0E:IAP2, 0x0F:AG, 0x10:AIO_CLIENT_SERVER,
             0x11:AVRC_CONTROLL, 0x12:AMS, 0x20:OPS, 0xFF:MISC, -1:UNKNOWN}


################################################################################

respond_to = {(0x20,0x04):"19 01 20 02 00 01 00"}


class Interpreter:
```

```
def __init__(self):
    self.table = rx_groups
    self.response_table = respond_to
    self.parsed_packets = 0
    self.new_packet()

#reset the stored packet info
def new_packet(self):
    self.group = -1
    self.event_num = -1
    self.length = -1
    self.params = []

#probably freaks out if there's an issue
#assumes it is a WICED header
#returns remaining length of packet
def parse_header(self, header_bytes):
    if header_bytes[0] is not 0x19:
        print "something is very wrong with the header"
        print header_bytes
        return -1

    self.event_num = header_bytes[1]
    self.group = header_bytes[2]
    self.length = header_bytes[3] + ( header_bytes[4] << 8 )
    if self.length > 50:
        print "length requirement of %s bytes is large" % self.length

    return self.length


def parse_params(self, param_bytes):
    try:
        param_format = self.table[self.group][self.event_num]
    except Exception:
        print Exception
```

```
            print "group: ", self.group
            self.new_packet()
            return
        for field_length in param_format:
            self.params.append(param_bytes[:field_length])
            param_bytes = param_bytes[field_length:]
        #variable length fields
        if len(param_bytes):
            self.params.append(param_bytes)
        self.parsed_packets += 1

    def is_HCI_packet(self):
        return self.group == 0 and self.event_num == 3

    def get_HCI_direction(self):
        return self.params[0][0]

    def get_HCI_packet_bytes(self):
        return self.params[1]

    def total_packets(self):
        return self.parsed_packets

    def hexify(self, p):
        q = lambda x: [hex(y)[2:].zfill(2) for y in x]
        return [q(x) for x in p]

    def response_required(self):
        return (self.group, self.event_num) in self.response_table.keys()

    def get_required_response(self):
        return self.response_table[(self.group, self.event_num)]

    def __str__(self):
        if self.is_HCI_packet():
            return "packet %s (HCI)" % self.total_packets()
```

```
        out = "packet %s (Cypress)\n" % self.total_packets()

        out += "group: %s\n" % self.group
        out += "event: %s\n" % self.event_num
        out += "length: %s\n" % self.length
        out += "parameters: %s" % self.hexify(self.params)

        return out
```

A.4 Plotter Module

```
import matplotlib.pyplot as plt

def show_throughput_plot(time_offsets, packet_sizes, file_name, log_timestamp):
    #format:
    #time offsets started at beginning of file, 0 bit packet size
    number_of_seconds = int(time_offsets[-1]) + 1
    binned_sizes = [0.0]*number_of_seconds
    times = range(number_of_seconds)
    for time, size in zip(time_offsets, packet_sizes):
        binned_sizes[int(time)] += round(size / 125.0, 4) #8/1000, bytes to kb

    average = sum(binned_sizes) / number_of_seconds
    #plot line first so the curve is on top
    plt.axhline(average, color='r')
    plt.plot(times, binned_sizes)
    plt.ylabel("kbps")
    plt.xlabel("seconds")
    plt.title(file_name)
    #need to save before show, not sure why
    plt.savefig("./files/%s_%s.png" % (log_timestamp, file_name[:-4]))
    #I believe show is blocking, don't want to leave it up indefinitely
    #plt.show()
    plt.close()
```

A.5 BTSnoop Logger Module

```
import time

def get_timestamp():
    t = time.time()*1000000 #seconds to useconds
    t += 66463223296000000
    #Magic number derived from testing. No idea where it comes from
    #fixes timestamp numbering in wireshark
    #it's over 2100 years
    return int(t)


class Logger:

    #need to add non-hci logging as a second file

    def __init__(self):
        self.num_HCI_packets = 0
        timestamp = get_timestamp()
        #print timestamp
        self.hci_log = open("./files/%s_btsnoop_log.bts" % timestamp, "wb")
        #.bts may not be correct, but I figured it was descriptive enough for
            now
        #some systems get angry if files don't have extensions
        #timestamp included so files aren't overwritten
        #need to have it opened as binary to fix accidentally writing CRLF
        self.hci_log.write(bytearray.fromhex("6274736E6F6F7000 00000001 000003E9
            "))
        #btsnoop file
        #file definition version 1
        #raw HCI format

    def good_hex(self, x):
        #you could replace lstrip with [2:] but that's less readable
```

```
        return hex(x).rstrip("L").lstrip("0x").zfill(2)

    #don't be a moron here
    def log_int_to_file(self, num, pad_length_bits):
        byte_character_length = pad_length_bits / 4
        #number of characters the string should have, 4 bits per character
        #turn int into hex string
        num_string = self.good_hex(num).zfill(byte_character_length)
        #python appends "L" to longs when you print them
        #remarkably, L is not a number in hex
        #pad to correct length
        chrs = [chr(int(''.join(x),16)) for x in zip(*[iter(num_string)]*2)]
        #see zip documentation
        for ch in chrs:
            self.hci_log.write(ch)

    #packet bytes is an array of ints in 0-255
    def log_HCI_packet(self, direction, packet_bytes):
        #number of bytes in packet
        self.log_int_to_file(len(packet_bytes), 32)
        #number of bytes from packet written in file
        self.log_int_to_file(len(packet_bytes), 32)
        #direction of the packet
        #direction pulled from the cypress packet
        self.log_int_to_file(3 - direction, 32)
        #cumulative packets missed
        #not planning on missing any
        self.log_int_to_file(0, 32)
        #timestamp
        self.log_int_to_file(get_timestamp(), 64)
        #packet
        for x in packet_bytes:
            self.hci_log.write(chr(x))
        self.num_HCI_packets += 1

    def close_logs(self):
        self.hci_log.close()
```

```
    def open_file_transfer(self, file_name):
        self.file = open("./files/%s" % file_name, "wb")

    def write_to_file_transfer(self, data_bytes):
        for x in data_bytes:
            self.file.write(chr(x))

    def close_file_transfer(self):
        self.file.close()
```

A.6 PCAP Logger Module

```
import time

def get_timestamp():
    t = time.time()*1000000 #seconds to useconds
    t += 66463223296000000
    #Magic number derived from testing. No idea where it comes from
    #fixes timestamp numbering in wireshark
    #it's over 2100 years
    return int(t)

def get_pcap_timestamp():
    t = time.time()
    t_us = (t - int(t)) * 1000000
    return int (t), int(t_us)


class Logger:

    #need to add non-hci logging as a second file

    def __init__(self):
        self.num_HCI_packets = 0
```

```
    timestamp = get_timestamp()
    #print timestamp
    self.hci_log = open("./snoop_logs/%s_pcaptest.pcap" % timestamp, "wb")
    #.bts may not be correct, but I figured it was descriptive enough for
        now
    #some systems get angry if files don't have extensions
    #timestamp included so files aren't overwritten
    #need to have it opened as binary to fix accidentally writing CRLF
    # self.hci_log.write(bytearray.fromhex("6274736E6F6F7000 00000001 000003
        E9"))
                                        #Decimal    Hex
    #Bluetooth_HCI_H4                      187      BB
    #Bluetooth_HCI_H4_w/PHDR               201      C9
    #BLE_LL                                251      FB
    #BLE_Linux_Monitor                     254      FE
    #BLE_BREDR_BB                          255      FF
    #BLE_LL_w/PHDR                         256      100  (00 01) in little
        endian
    #Default (Ethernet)                    1        1
    #Little Endian
    self.hci_log.write(bytearray.fromhex("a1 b2 c3 d4 00 02 00 04 00 00 00
        00 00 00 00 00 00 00 ff ff 00 00 00 c9"))

    #btsnoop file
    #file definition version 1
    #raw HCI format

    self.serial_log = open("./snoop_logs/%s_serial_log.txt" % timestamp, "wb
        ")

def good_hex(self, x):
    #you could replace lstrip with [2:] but that's less readable
    return hex(x).rstrip("L").lstrip("0x").zfill(2)

#don't be a moron here
def log_int_to_file(self, num, pad_length_bits):
    byte_character_length = pad_length_bits / 4
```

```
        #number of characters the string should have, 4 bits per character
        #turn int into hex string
        num_string = self.good_hex(num).zfill(byte_character_length)
        #python appends "L" to longs when you print them
        #remarkably, L is not a number in hex
        #pad to correct length
        chrs = [chr(int(''.join(x),16)) for x in zip(*[iter(num_string)]*2)]
        #see zip documentation
        for ch in chrs:
            self.hci_log.write(ch)

    #packet bytes is an array of ints in 0-255
    def log_HCI_packet(self, direction, packet_bytes):
        #PCAP timestamp
        ts_sec, ts_u_sec = get_pcap_timestamp()
        #ts_sec - Seconds only epoch time
        self.log_int_to_file(ts_sec,32)
        #ts_u_usec - Microseconds offset to ts_sec
        self.log_int_to_file(ts_u_sec, 32)
        #PCAP length
        #incl_len
        self.log_int_to_file(len(packet_bytes) + 1, 32)
        #orig_len
        self.log_int_to_file(len(packet_bytes) + 1, 32)
        # #number of bytes in packet
        # self.log_int_to_file(len(packet_bytes), 32)
        # #number of bytes from packet written in file
        # self.log_int_to_file(len(packet_bytes), 32)
        # #direction of the packet
        # #direction pulled from the cypress packet
        # self.log_int_to_file(3 - direction, 32)
        # #cumulative packets missed
        # #not planning on missing any
        # self.log_int_to_file(0, 32)
        # #timestamp
        # self.log_int_to_file(get_timestamp(), 64)
        #packet
```

```
        testing = [direction] + packet_bytes
        print testing
        for x in testing:
            self.hci_log.write(chr(x))
        self.num_HCI_packets += 1
        self.serial_log.write("HCI packet %s\n" % self.num_HCI_packets)
        #self.log_serial_packet(packet_bytes)

    def log_serial_packet(self, packet):
        p = " ".join([self.good_hex(x) for x in packet])
        self.serial_log.write(p)
        self.serial_log.write("\n\n")

    def log_serial_flush(self, data):
        self.serial_log.write("error, flushing\n")
        p = " ".join([self.good_hex(x) for x in data])
        self.serial_log.write(p)
        self.serial_log.write("\n\n")


    def log_outgoing_packet(self, packet_string):
        self.serial_log.write("TX: %s\n\n" % packet_string)

    def close_logs(self):
        self.hci_log.close()
        self.serial_log.close()
```

A.7 PCAPNG Logger Module

```
import time

def get_timestamp():
    t = time.time()*1000000 #seconds to useconds
    t += 66463223296000000
    #Magic number derived from testing. No idea where it comes from
```

```
    #fixes timestamp numbering in wireshark
    #it's over 2100 years
    return int(t)

def get_pcap_timestamp():
    t = time.time()
    t_us = (t - int(t)) * 1000000
    return int (t), int(t_us)

    #for microseconds resolution only
def get_pcapng_timestamp():
    t = time.time()
    t_us = t * 1000000
    t_high = int(t_us) >> 32
    t_low = int(t_us) & 0x00000000ffffffff
    return int(t_high), int(t_low)

class Logger:

    def __init__(self):
        self.num_HCI_packets = 0
        self.timestamp = get_pcap_timestamp()[0]
        #print timestamp

    def good_hex(self, x):
        #you could replace lstrip with [2:] but that's less readable
        return hex(x).rstrip("L").lstrip("0x").zfill(2)

    #don't be a moron here
    def log_int_to_file(self, num, pad_length_bits):
        byte_character_length = pad_length_bits / 4
        #number of characters the string should have, 4 bits per character
        #turn int into hex string
        num_string = self.good_hex(num).zfill(byte_character_length)
        #python appends "L" to longs when you print them
        #remarkably, L is not a number in hex
        #pad to correct length
```

```
        chrs = [chr(int(''.join(x),16)) for x in zip(*[iter(num_string)]*2)]
        #see zip documentation
        self.hci_log.write(''.join(chrs))

    #Wireshark doesn't recognize all packet types for some reason
    #packet bytes is an array of ints in 0-255
    def log_HCI_packet(self, direction, packet_bytes):
        #Block Type (Block type number for EPB is 6)
        self.hci_log.write(bytearray.fromhex("0000 0006"))

        #Padding calculation to comply with 32 bit data allignment
        padding = 4 - ((abs((len(packet_bytes) + 1 + 4) - 4)) %4) %4

        #Block Total Length
        total_block_length = len(packet_bytes) + 1 + 4 + padding + 32
        self.log_int_to_file(total_block_length, 32)
        #Interface ID (This field is zero since we are using one interface can
            be change later)
        self.hci_log.write(bytearray.fromhex("0000 0000"))

        #PCAPNG Timestamp (64 bit - microseconds resolution)
        # t = time.time()
        # t_us = t * 1000000
        # t_us_int = int(t_us)
        # print hex(t_us_int).rstrip("L").lstrip("0x").zfill(16)
        ts_us_high, ts_us_low = get_pcapng_timestamp()
        #Timestamp (Higher 32 bits)
        self.log_int_to_file(ts_us_high,32)
        #Timestamp (Lower 32 bits)
        self.log_int_to_file(ts_us_low,32)

        #Packet Length
        direction_of_packet = direction % 2
        #Captured Packet length
        self.log_int_to_file(len(packet_bytes) + 1 + 4, 32)
        #Original Packet Length (In reality the captured & original length
            should be the same )
```

```
        self.log_int_to_file(len(packet_bytes) + 1 + 4, 32)
        #Direction of packet field
        self.log_int_to_file(direction_of_packet, 32)
        #Packet
        #trying to fix h4 formatting with the below
        packet_indicator = [4,1]
        if direction > 1:
             ACL_intmd = packet_bytes.encode("hex")[4:8]
             ACL_trans = ACL_intmd[2] + ACL_intmd[3] + ACL_intmd[0] + ACL_intmd
                 [1]
             if int(ACL_trans,16) == ((len(packet_bytes.encode("hex")[8:]))/2):#
                 ACL
                self.log_int_to_file(2,8)
             elif int(packet_bytes.encode("hex")[4:6],16) == ((len(
                 packet_bytes.encode("hex")[6:]))/2):#SCO
                self.log_int_to_file(3,8)
             else: #this should never happen
                self.log_int_to_file(2,8)
        else:
            self.log_int_to_file(packet_indicator[direction],8)
        self.hci_log.write(packet_bytes)
        # Padding
        for x in range(padding):
           self.hci_log.write(bytearray.fromhex("00"))
        #BLock Total length
        self.log_int_to_file(total_block_length, 32)

        self.num_HCI_packets += 1

    def close_hci_log(self):
        self.hci_log.close()

    def open_file_transfer(self, file_name):
        self.file = open("./files/%s_%s" % (self.timestamp, file_name), "wb")

    def open_hci_log(self):
        self.hci_log = open("./files/%s_hci_log.pcapng" % self.timestamp, "wb")
```

```
        #some systems get angry if files don't have extensions
        #timestamp included so files aren't overwritten
        #need to have it opened as binary to fix accidentally writing CRLF
        # self.hci_log.write(bytearray.fromhex("a1b2c3d4 0002 0004 00000000
            00000000 0000ffff 000000c9"))
        self.hci_log.write(bytearray.fromhex("0a0d0d0a 0000001c 1a2b3c4d 0001
            0000 FFFFFFFFFFFFFFFF 0000 001c "))
        '''PCAP Global Header    | PCAPNG Section Header Block    | Length (8 bit
            = 1 octet, 32 bit = 4 octet)
        Magic Number             | Block Type                     | 4 octet
        Major Version            | Block Total Length             | 4 octet
        Minor Version            | Magic Number                   | 4 octet
        Timezone                 | Major Version                  | 2 octet
        Sig Figs                 | Minor Version                  | 2 octet
        Max Packet Size          | Section Length                 | 8 octet
        Data Link Type           | Options                        | 0 octet (
            variable)
                                 | Block Total Length             | 4 octet
                                 | Total                          | 28 octet
        '''
        self.hci_log.write(bytearray.fromhex("00000001 00000014 00c9 0000
            00000000 00000014"))
        '''PCAPNG Interface Description Block   |Length
         Block Type                             | 4 octet
         BLock Total Length                     | 4 octet
         Link Type                              | 2 octet
         Reserved                               | 2 octet
         SnapLen                                | 4 octet
         Options                                | 0 octet (variable)
         Block Total Length                     | 4 octet
         Total                                  | 32 octet
         '''

    def write_to_file_transfer(self, data_bytes):
        self.file.write(data_bytes)

    def close_file_transfer(self):
```

```
        self.file.close()
```

Appendix B

Litepoint-based Promiscuous Packet Capture Tool

B.1 Litepoint Packet to JSON Translator

```
# coding: utf-8
'''
Instructs a Litepoint IQxel to re-analyze all
packets within an IQ capture in Channel 1 and write results
to a JSON file.
'''


import sys
import lime
import json
import time
import datetime
import argparse


def help():
```

```
    print("Usage: ")
    print("    python %s [<jsonFile>]" % __file__)
    print("    [<jsonFile>] - filename for JSON packet storage")
    exit()


def packet_to_json(jsonFilename, LP_IP_address='169.254.22.24'):
    '''
    dictofPacketsDefault = {
        "Capture Timestamp": "01:01:1980:13:00:00.000000",
        "Channel Index": 0,
        "Detected IF Bandwidth": 0,
        "Detected Payload Length (bytes)": 0,
        "Detected Data Rate": 0,
        "CRC Pass?": False,
        "Preamble Bytes": "",
        "Header Bytes": [
                ""
        ],
        "Payload Bytes": "",
        "Signal Bandwidth (MHz)": 0
    }'''

    dictofPacketsDefault = {
        "Capture Timestamp": "01:01:1980:13:00:00.000000",
        "Preamble Bytes": "",
        "Header Bytes": [
                ""
        ],
        "Payload Bytes": "",
        "CRC Bytes": ""
    }
    dictofPackets = {}
    dictofAllPackets = {}

    lime.initLime()
    con = lime.connect(address=LP_IP_address, port=24000)
```

```
con.setTimeout(60000)


lime.Print("Getting packet count:\n")
scpi_commands = '''
    BT;CONF:DRAT LEN;
    BT;CONF:DEWH ON;
    BT;CONF:CHAN:AUTO ON;
    BT;CONF:SLOC CIND;
    BT;CONF:LEN:SWOR:AUTO ON;
    BT;CONF:LEN:PHE LINK;
    BT
    CHAN1
    BT
    CALC:POW 0,1
    BT;FETC:SYNC?
    '''
r = con.query1d(scpi_commands)
if r[0]=='0':
    packet_count = int(r[2])
    print("packet_count : %d" % packet_count)

    ts = time.time()
    capturetime = datetime.datetime.fromtimestamp(ts).strftime('%m:%d:%Y:%H
        :%M:%S.%f')

    for packetNumber in range(0,packet_count):
        try:
            print("")
            print("*****Packet : %d" % (packetNumber+1))

            #get packet type here!!
            #determine analysis type


            scpi_commands_template = '''
                CHAN1
```

```
    BT
    CLE:ALL
    CALC:POW {0},1
    CALC:TXQ {0},1
    CALC:SPEC {0},1
    '''
scpi_commands = scpi_commands_template.format(packetNumber, 1)
print(scpi_commands)
con.scpi_exec(scpi_commands)

r = con.query1d("*wai;err:all?")
if int(r[0]) != 0:
    raise Exception("{} ran into errors: {}".format(
        scpi_commands, r))

dictofPackets = dictofPacketsDefault.copy()

dictofPackets['Capture Timestamp'] = capturetime

'''
#Channel index
scpi_commands =
    BT;FETC:SEGM1:TXQ:CHAN?

#print(scpi_commands)
r = con.query1d(scpi_commands)
dictofPacket['Channel Index'] = r[1]
'''
#IF bandwdith

#Signal bandwidth

#Data rate

#CRC pass?

#Preamble Bytes
```

```
scpi_commands = '''
    BT;FETC:SEGM1:TXQ:LEN:PRE?
    '''
#print(scpi_commands)
r = con.query1d(scpi_commands)
if r[0]=='0' :
    bits = r[1]
    packet_hex = ''
    print("Number of bits: %d" % len(bits))
    for byte in range(0,len(bits),8):
        byte_int=0
        for b in range(0,8):
            if bits[byte+b] == 0x01:
                byte_int += 2**b
        packet_hex += "{:02x}".format(byte_int)

    dictofPackets['Preamble Bytes'] = packet_hex
else :
    print("BITS query error")

#Header Bytes
scpi_commands = '''
    BT;FETC:SEGM1:TXQ:LEN:PDUH?
    '''
#print(scpi_commands)
r = con.query1d(scpi_commands)

if r[0]=='0' :
    bits = r[1]
    packet_hex = ''
    print("Number of bits: %d" % len(bits))
    for byte in range(0,len(bits),8):
        byte_int=0
        for b in range(0,8):
            if bits[byte+b] == 0x01:
                byte_int += 2**b
        packet_hex += "{:02x}".format(byte_int)
```

```
    dictofPackets['Header Bytes'] = packet_hex
else :
    print("BITS query error")

#Payload bits
scpi_commands = '''
    BT;FETC:SEGM1:TXQ:CLAS:PAYL?
    '''
print(scpi_commands)
r = con.query1d(scpi_commands)
#print(r)
if r[0]=='0' :
    bits = r[1]
    packet_hex = ''
    print("Number of bits: %d" % len(bits))
    for byte in range(0,len(bits),8):
        byte_int=0
        for b in range(0,8):
            if bits[byte+b] == 0x01:
                byte_int += 2**b
        packet_hex += "{:02x}".format(byte_int)

    dictofPackets['Payload Bytes'] = packet_hex
else :
    print("BITS query error")

#CRC bytes
scpi_commands = '''
    BT;FETC:SEGM1:TXQ:LEN:CRC?
    '''
print(scpi_commands)
r = con.query1d(scpi_commands)
#print(r)
if r[0]=='0' :
    bits = r[1]
    packet_hex = ''
```

```
                    print("Number of bits: %d" % len(bits))
                    for byte in range(0,len(bits),8):
                        byte_int=0
                        for b in range(0,8):
                            if bits[byte+b] == 0x01:
                                byte_int += 2**b
                        packet_hex += "{:02x}".format(byte_int)

                    dictofPackets['CRC Bytes'] = packet_hex
                else :
                    print("CRC query error")
                key = str(int(packetNumber)+1)
                dictofAllPackets.update({key:dictofPackets})
            except:
                pass

        print(dictofAllPackets)
        json_str = json.dumps(dictofAllPackets, indent=4)

        print(json_str)
        f = open(jsonFilename,"w")
        f.write(json_str)
        f.close()

    else:
        lime.error("No valid packets found\n")



if __name__ == "__main__":

    parser = argparse.ArgumentParser(description='This script commands a
        LitePoint analyzer to re-analyze and then save results to JSON.')
    parser.add_argument('LP_IP_address', type=str, help='IP address of the
        LitePoint analyzer')
    parser.add_argument('JSON_file', type=str, help='Name of output JSON file')
    args = parser.parse_args()
```

```
    LP_IP_address = args.LP_IP_address
    jsonFile = args.JSON_file

    # Read the parameters
    '''
    if len(sys.argv) >= 3:
        help()

    if len(sys.argv) <= 1:
        help()
    jsonFile           = sys.argv[1]
'''

    print("jsonFile         =%s " % jsonFile)

    packet_to_json(jsonFile, LP_IP_address=LP_IP_address)
```

B.2 JSON to PCAP Translator

```
import os
import subprocess
import sys
import json
import time
import datetime
import binascii

if (sys.version_info > (3, 0)):
    print("Python 3 not yet supported")
    sys.exit()

#Need to install hexdump package
import hexdump

def byte_to_int(x) :
```

```
    x = int(binascii.hexlify(x),16)
    return x

def crc_check(input_bitstring, polynomial_bitstring, check_value):
    '''
    Calculates the CRC check of a string of bits using a chosen polynomial.
    initial_filler is assumed to be '1'.
    '''
    len_input = len(input_bitstring)
    initial_padding = check_value
    input_padded_array = list(input_bitstring + initial_padding)
    polynomial_bitstring = polynomial_bitstring.lstrip('0')
    while '1' in input_padded_array[:len_input]:
        cur_shift = input_padded_array.index('1')
        for i in range(len(polynomial_bitstring)):
            if polynomial_bitstring[i] == input_padded_array[cur_shift + i]:
                input_padded_array[cur_shift + i] = '0'
            else:
                input_padded_array[cur_shift + i] = '1'
    if '0' not in ''.join(input_padded_array)[len_input:]:
        return True
    else:
        return False


def help():
    print("Usage: ")
    print("    python %s [<jsonFile>] [<pcapFile>]" % __file__)
    print("    [<jsonFile>] - filename for JSON packet storage")
    print("    [<pcapFile>] - filename for PCAP packet storage")
    exit()


jsonFile = ''
pcapFile = ''
writeAMPDU = 0
iqSourceFile = ''
```

```
# Read the parameters
if len(sys.argv) >= 4:
    help()

if len(sys.argv) <= 1:
    help()

jsonFile           = sys.argv[1]
pcapFile           = sys.argv[2]

print("jsonFile         =%s " % jsonFile)
print("pcapFile         =%s " % pcapFile)

with open(jsonFile) as f:
    dictofPackets = json.load(f)


pcap_file = open("packets.txt", 'w')


for key in sorted(dictofPackets, key=int):

    packet = dictofPackets[key]
    packethex = '00000000' + packet['Header Bytes'] + packet['Payload Bytes'] +
        packet['CRC Bytes']

    if (sys.version_info > (3, 0)):
        print("TBD")
    else :
        packetbytes = packethex.decode("hex")
    capturets = time.mktime(datetime.datetime.strptime(packet['Capture Timestamp
        '], '%m:%d:%Y:%H:%M:%S.%f').timetuple())
    packetts = capturets #+ float(packet['Packet Start Time (us)']) / 1000000
    packettime = datetime.datetime.fromtimestamp(packetts).strftime("%m:%d:%y:%H
        :%M:%S.%f")
    pcap_file.write(packettime)
```

```
    pcap_file.write('\n')
    old_stdout = sys.stdout
    sys.stdout = pcap_file
    hexdump.hexdump(packetbytes)
    sys.stdout = old_stdout



pcap_file.close()

text2pcap_path  = r"C:\Program Files\Wireshark-octoScope\text2pcap.exe"

cmd = [ text2pcap_path, '-l', '251', 'packets.txt', pcapFile, '-t', '%m:%d:%y:%H
    :%M:%S.' ]
print ("cmd: %s" % cmd)
subprocess.call(cmd)
```