



**NEAR REAL-TIME RF-DNA FINGERPRINTING FOR
ZIGBEE DEVICES USING SOFTWARE DEFINED
RADIOS**

THESIS

Frankie A. Cruz, Captain, USAF
AFIT-ENG-MS-19-M-021

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views expressed in this document are those of the author and do not reflect the official policy or position of the United States Air Force, the United States Department of Defense or the United States Government. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

AFIT-ENG-MS-19-M-021

NEAR REAL-TIME RF-DNA FINGERPRINTING FOR ZIGBEE DEVICES USING
SOFTWARE DEFINED RADIOS

THESIS

Presented to the Faculty
Department of Electrical Engineering
Graduate School of Engineering and Management
Air Force Institute of Technology
Air University
Air Education and Training Command
in Partial Fulfillment of the Requirements for the
Degree of Master of Science in Computer Engineering

Frankie A. Cruz, B.S.E.E.

Captain, USAF

March 21, 2019

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT-ENG-MS-19-M-021

NEAR REAL-TIME RF-DNA FINGERPRINTING FOR ZIGBEE DEVICES USING
SOFTWARE DEFINED RADIOS

THESIS

Frankie A. Cruz, B.S.E.E.
Captain, USAF

Committee Membership:

Maj J. Addison Betances, PhD
Chair

Dr. Michael A. Temple
Member

Dr. Timothy H. Lacey
Member

Abstract

Low-Rate Wireless Personal Area Network(s) (LR-WPAN) usage has increased as more consumers embrace Internet of Things (IoT) devices. ZigBee Physical Layer (PHY) is based on the Institute of Electrical and Electronics Engineers (IEEE) 802.15.4 specification designed to provide a low-cost, low-power, and low-complexity solution for Wireless Sensor Network(s) (WSN). The standard's extended battery life and reliability makes ZigBee WSN a popular choice for home automation, transportation, traffic management, Industrial Control Systems (ICS), and cyber-physical systems. As robust and versatile as the standard is, ZigBee remains vulnerable to a myriad of common network attacks. Previous research involving Radio Frequency-Distinct Native Attribute (RF-DNA) Fingerprinting and device discrimination has shown that bit-level WSN security can be augmented with PHY-based features. The objective of this research was to develop and implement an Radio Frequency (RF) air monitor system that classifies devices in Near Real-Time (NRT). The performance of the NRT air monitor is contrasted against previous research that utilized MATLAB-based Fingerprinting post-processing RF-DNA.

The RF air monitor demonstration included collection of IEEE 802.15.4 bursts from $N_d = 10$ RZUSBsticks to assess NRT performance and effectiveness. The first set of experiments examined how well the air monitor recovered IEEE 802.15.4 data packets while fingerprinting and discriminating ZigBee devices under two distinct workloads. The second set of experiments compared predictive post-processed MATLAB RF-DNA Multiple Discriminant Analysis/Maximum Likelihood (MDA/ML) models Average Percent Correct Classification (%C) against the air monitor's observed operational %C for each RZUSBstick. The air monitor achieved an Overall Accurate Packet Reconstruction

Percent ($\%R$) $\geq 97.92\%$ while correctly fingerprinting an Overall Fingerprinted Percent ($\%F$) $\geq 97.48\%$ of the transmitted IEEE 802.15.4 data packets during the trials. The air monitor achieved an overall operational $\%C \approx 96.93\%$ at a collected Signal-to-Noise Ratio (SNR) ≈ 33.571 dB, classified each RZUSBstick within $0.45 \text{ msec} \leq T_{\text{MDA}} \leq 1.5 \text{ msec}$ after detection, and $\%C$ Deviation ($\%C_{\Delta}$) $= 2.71\%$ from the collected post-processed MDA/ML model. The results support that an RF air monitor is feasible, can be effective, and will accurately operate within predictive post-processed MATLAB model estimations.

Acknowledgements

I would like to thank my wife and children, I wouldn't be where I am today without thier love and support. I would also like to thank Major Betances for his guidance and mentoring throught my tenure at AFIT.

Frankie A. Cruz

Table of Contents

	Page
Abstract	iv
Acknowledgements	vi
List of Figures	x
List of Tables	xiii
List of Abbreviations	xv
I. Introduction	1
1.1 Operational Motivation	1
1.2 Technical Motivation	2
1.3 Document Organization	2
II. Background.....	4
2.1 ZigBee.....	4
2.2 Offset-Quadrature Phase Shift Keying (O-QPSK) Modulation	9
2.3 Minimum Shift Keying (MSK) Modulation.....	10
2.4 Radio Frequency-Distinct Native Attribute (RF-DNA) Fingerprinting	11
2.5 Multiple Discriminant Analysis/Maximum Likelihood (MDA/ML)	13
2.6 Ramsey STE4400.....	16
2.7 KillerBee: ZigBee Attack Platform	16
2.8 Wireshark	16
2.9 Cyclic Redundancy Check (CRC)-16	17
2.10 Eigen	17
2.11 Software-Defined Radio(s) (SDR)	18
2.12 GNU Radio.....	18
III. Methodology	20
3.1 GNU Radio SDR Receiver Design	21
3.1.1 <i>Institute of Electrical and Electronics Engineers (IEEE) 802.15.4 Physical Layer (PHY)</i> Transceiver Block.....	23
3.1.2 RF-DNA Air Monitor	24

3.1.2.1	Universal Software Radio Peripheral(s) (USRP) Receive (RX) Interface	24
3.1.2.2	Demodulation and Detection	27
3.1.2.3	Sample Delay	29
3.1.2.4	Burst Capture and RF-DNA Processing	30
3.1.2.5	Data Visualization	36
3.1.2.6	Data Storage	38
3.2	Burst Detection	39
3.3	Fingerprint Generation.....	42
3.3.1	MATLAB Fingerprint Generation	43
3.3.2	GNU Radio Fingerprint Generation	47
3.4	MDA/ML	48
IV.	Analysis of Results	51
4.1	GNU Radio Air Monitor Collection Performance Analysis	51
4.2	GNU Radio Air Monitor Operational Performance Analysis.....	58
4.3	Air Monitor MDA/ML Classification Model Development	60
4.4	Near Real-Time (NRT) Device Discrimination Analysis	61
V.	Conclusion	64
5.1	Results Summary	64
5.2	Research Contribution	64
5.3	Future Research Recommendation	67
Appendix A.	Device Collection Figures	69
Appendix B.	Device Collection Tables	85
Appendix C.	Statistical File Examples	91
C.1	preamble_sink Statistical File Example	91
C.2	Dna_detector_ccf Device Statistical File Example	92
C.3	Dna_detector_ccf Fingerprint Statistical File Example	93
Appendix D.	Source Code	98
D.1	Packet Sink Block Source Code	98
D.1.1	Packet Sink C++ Code.....	98
D.1.2	Packet Sink XML GRC Code	122
D.2	Dna detector ccf Block Source Code	125
D.2.1	Dna detector ccf C++ Code	125
D.2.2	Dna detector ccf XML GRC Code	147
D.3	DNA Buffer Class	150
D.4	Base Packet and IEEE 802.15.4 Packet Classes	156
D.5	File Read Implementation Class	165

D.6 File Store Base and File Store Implementation Classes	170
Bibliography.....	175

List of Figures

Figure	Page
2.1	Frequencies and Channel Spread for ZigBee in the 2450.0 MHz Band [1] 5
2.2	Flowchart of ZigBee's Direct Sequence Spread Spectrum (DSSS) Implementation [1] 8
2.3	PPDU Frame Format [1, 9, 23] 8
2.4	O-QPSK In-Phase and Quadrature-Phase (I/Q) Waveform with Time Offset T_d [23, 25] 10
2.5	Quadrature Phase Shift Keying (QPSK) Phase Transition Diagram [26] 10
2.6	O-QPSK Phase Transition Diagram [26] 11
2.7	Summarized Overview of RF-DNA Time Domain (TD) Fingerprinting [14, 27] 14
2.8	Representative Multiple Discriminant Analysis (MDA) Projections using W_1 and W_2 for a $N_C = 3$ problem into a $(N_C - 1)$ -Dimensional Space [27]. 15
3.1	Methodology for NRT RF-DNA Fingerprinting and Device Discrimination 20
3.2	Air Monitor Test Configuration 22
3.3	<i>IEEE O-QPSK 802.15.4</i> Transceiver Block..... 23
3.4	<i>IEEE 802.15.4 PHY</i> Transceiver Block Flow Graph..... 24
3.5	Complete GNU Radio Companion RF-DNA Air Monitor Flow Graph..... 25
3.6	USRP RX Interface Blocks..... 26
3.7	Demodulation and Detection Blocks..... 27
3.8	GNU Radio Clock Recovery Mueller and Müller (M&M) Illustration 28
3.9	Sample Delay Blocks 29

3.10	Burst Capture and RF-DNA Processing Blocks	30
3.11	Finite State Machine Diagram for <i>preamble_sink</i>	33
3.12	<i>Dna detector ccf</i> Block State Diagram	37
3.13	Data Visualization Blocks	38
3.14	Air Monitor Operational Graphical User Interface (GUI)	38
3.15	Data Storage Blocks	39
3.16	Normalized IEEE 802.15.4 Packet and Synchronization Header (SHR) Bursts from Device 10	41
3.17	Detected IEEE 802.15.4 SHR Burst from Device 10	42
3.18	Normalized Burst at $R_{\text{Samp}} = 10$ MSamp/sec from Device 10	45
3.19	Normalized Burst at $R_{\text{Samp}} = 20$ MSamp/sec from Device 10	45
3.20	Filtered Normalized Burst at $R_{\text{Samp}} = 20$ MSamp/sec from Device 10	46
3.21	Normalized Burst at $R_{\text{Samp}} = 4$ MSamp/sec from Device 10	46
3.22	$N_{\text{Bursts}} = 239$ Bursts from <i>control4-sample.pcap</i> on Device 10	48
A-1	I/Q Plot on Device 1 of $N_{\text{Bursts}} = 239$ bursts from Libpcap File	69
A-2	I/Q Plot on Device 2 of $N_{\text{Bursts}} = 239$ bursts from Libpcap File	69
A-3	I/Q Plot on Device 3 of $N_{\text{Bursts}} = 239$ bursts from Libpcap File	70
A-4	I/Q Plot on Device 4 of $N_{\text{Bursts}} = 239$ bursts from Libpcap File	70
A-5	I/Q Plot on Device 5 of $N_{\text{Bursts}} = 239$ bursts from Libpcap File	71
A-6	I/Q Plot on Device 6 of $N_{\text{Bursts}} = 239$ bursts from Libpcap File	71
A-7	I/Q Plot on Device 7 of $N_{\text{Bursts}} = 239$ bursts from Libpcap File	72
A-8	I/Q Plot on Device 8 of $N_{\text{Bursts}} = 239$ bursts from Libpcap File	72
A-9	I/Q Plot on Device 9 of $N_{\text{Bursts}} = 239$ bursts from Libpcap File	73
A-10	I/Q Plot on Device 10 of $N_{\text{Bursts}} = 239$ bursts from Libpcap File	73

A-11	(a) I/Q and (b) Power Plot on Device 1 of $N_{\text{Bursts}} = 1,912$ bursts from Libpcap File Replayed 8-Times	74
A-12	(a) I/Q and (b) Power Plot on Device 1 of $N_{\text{Bursts}} = 1,912$ bursts from Libpcap File Replayed 8-Times	75
A-13	(a) I/Q and (b) Power Plot on Device 2 of $N_{\text{Bursts}} = 1,912$ bursts from Libpcap File Replayed 8-Times	76
A-14	(a) I/Q and (b) Power Plot on Device 3 of $N_{\text{Bursts}} = 1,912$ bursts from Libpcap File Replayed 8-Times	77
A-15	(a) I/Q and (b) Power Plot on Device 4 of $N_{\text{Bursts}} = 1,912$ bursts from Libpcap File Replayed 8-Times	78
A-16	(a) I/Q and (b) Power Plot on Device 5 of $N_{\text{Bursts}} = 1,912$ bursts from Libpcap File Replayed 8-Times	79
A-17	(a) I/Q and (b) Power Plot on Device 6 of $N_{\text{Bursts}} = 1,912$ bursts from Libpcap File Replayed 8-Times	80
A-18	(a) I/Q and (b) Power Plot on Device 7 of $N_{\text{Bursts}} = 1,912$ bursts from Libpcap File Replayed 8-Times	81
A-19	(a) I/Q and (b) Power Plot on Device 8 of $N_{\text{Bursts}} = 1,912$ bursts from Libpcap File Replayed 8-Times	82
A-20	(a) I/Q and (b) Power Plot on Device 9 of $N_{\text{Bursts}} = 1,912$ bursts from Libpcap File Replayed 8-Times	83
A-21	(a) I/Q and (b) Power Plot on Device 10 of $N_{\text{Bursts}} = 1,912$ bursts from Libpcap File Replayed 8-Times	84

List of Tables

Table	Page
2.1 ZigBee Symbol-to-Chip Mapping for the O-QPSK 2450.0 MHz Band [20, 24]	7
2.2 ZigBee Symbol-to-Chip Mapping for the MSK 2450.0 MHz Band [24]	7
3.1 RZUSBstick's Media Access Control Layer (MAC) Address, AT86RF230 Transceiver Marks, Device Date Code, and Serial Number (SN)	22
4.1 Composition of <i>control4-sample.pcap</i> With and Without acknowledgement (ACK) Packets	53
4.2 Ideal Composition of $N_{\text{Total Packets}} = 19,120$ Detected Packets	55
4.3 Overall Collection Performance for $N_d = 10$ Devices	57
4.4 Overall Operational Performance for $N_d = 10$ Devices	59
4.5 Estimative RF-DNA MDA/ML Cross Classification Performance Model from $0 \text{ dB} \leq \text{Signal-to-Noise Ratio (SNR)} = 40 \text{ dB}$	61
4.6 Confusion Matrix $N_d = 10$ Class Problem at $\text{SNR} \approx 33.571 \text{ dB}$	61
4.7 Operational NRT Device Discrimination Raw Confusion Matrix (CM) Totals for $N_d = 10$ Devices	62
4.8 Operational NRT Device Discrimination Average Percent Correct Classification (%C) CM for $N_d = 10$ Devices	63
1 Collection Performance Tables for Devices 1 to 4	85
2 Collection Performance Tables for Devices 5 to 8	86
3 Collection Performance Tables for Devices 9 and 10	87
4 Operational Performance Tables for Devices 1 to 4	88
5 Operational Performance Tables for Devices 5 to 8	89
6 Operational Performance Tables for Devices 9 and 10	90

7	Operational NRT Device Discrimination Joint Probability	
	Mass Function (PMF) CM for $N_d=10$ Devices	90

List of Abbreviations

$\%A_{\Delta}$	Absolute Reconstructed Packet Ratio Deviation
AFIT	Air Force Institute of Technology
AWGN	Additive White Gaussian Noise
BPSK	Binary Phase Shift Keying
BPSK	Binary Phase Shift Keying
CB-DNA	Constellation Based-Distinct Native Attributes
CM	Confusion Matrix
CPM	Continuous Phase Modulation
CRC	Cyclic Redundancy Check
$\%C$	Average Percent Correct Classification
$\%C_{\Delta}$	$\%C$ Deviation
ACK	acknowledgement
DNA	Distinct Native Attribute
DDR3	Double Data Rate Type 3 Synchronous Dynamic Random-Access Memory
DDR3L	Low Voltage Double Data Rate Type 3 Synchronous Dynamic Random-Access Memory
DS	Data Symbol(s)
DSSS	Direct Sequence Spread Spectrum

DSP	Digital Signal Processing
FIFO	First-In-First-Out
GUI	Graphical User Interface
GPL	GNU General Public License
GPU	Graphics Processing Unit
ICS	Industrial Control Systems
IIR	Infinite Impulse Response
I	In-Phase
M&M	Mueller and Müller
IoT	Internet of Things
Q	Quadrature-Phase
I/Q	In-Phase and Quadrature-Phase
%I	Individual % Incorrect Classification
IEEE	Institute of Electrical and Electronics Engineers
ℑ	Imaginary
IIR	infinite impulse response
ITU-T	International Telecommunication Union - Telecommunication Standardization Sector
LR-WPAN	Low-Rate Wireless Personal Area Network(s)

MDA	Multiple Discriminant Analysis
MAC	Media Access Control Layer
ML	Maximum Likelihood
MDA/ML	Multiple Discriminant Analysis/Maximum Likelihood
MODEM	Modulator/Demodulator
MSK	Minimum Shift Keying
NRT	Near Real-Time
O-QPSK	Offset-Quadrature Phase Shift Keying
OOT	Out-of-Tree
OS	Operating System
PHY	Physical Layer
PHR	PHY Header
QPSK	Quadrature Phase Shift Keying
$\Re$	Real
ROI	Region of Interest
PMF	Probability Mass Function
$\%D$	Overall Packet Detection Percent
$\%R$	Overall Accurate Packet Reconstruction Percent
$\%F$	Overall Fingerprinted Percent

PPDU	PHY Protocol Data Unit
PSDU	PHY Service Data Unit
PRNG	Pseudo Random Noise Generated
PHR	PHY Header
RF	Radio Frequency
RX	Receive
RF-DNA	Radio Frequency-Distinct Native Attribute
ROI	Region(s) of Interest
SD	Spectral Domain
SNR	Signal-to-Noise Ratio
SDR	Software-Defined Radio(s)
SFD	Start-of-Frame Delimiter
SHR	Synchronization Header
SNA	Sensor Network Analyzer
SN	Serial Number
SSD	Solid State Drive
TD	Time Domain
USRP	Universal Software Radio Peripheral(s)
USB	Universal Serial Bus

uInt32 32-Bit Unsigned Integer

WPAN Wireless Personal Area Network(s)

WSN Wireless Sensor Network(s)

NEAR REAL-TIME RF-DNA FINGERPRINTING FOR ZIGBEE DEVICES USING SOFTWARE DEFINED RADIOS

I. Introduction

This chapter provides the operational and technical motivations for conducting this research. Section 1.1 describes the operational motivation for looking into the ZigBee protocol. Section 1.2 provides the technical motivation to fingerprint and discriminate ZigBee devices in Near Real-Time (NRT). Section 1.3 concludes Chapter I with the organizational details for the remainder of this document.

1.1 Operational Motivation

ZigBee is an Institute of Electrical and Electronics Engineers (IEEE) 802.15.4 based standard ideal for low-data rate applications that require an extended battery life and reliable networks. As such, ZigBee Wireless Personal Area Network(s) (WPAN) have gained popularity to the point where they have become the de facto standard for Wireless Sensor Network(s) (WSN) [1]. ZigBee network's low-cost, low-power, and low-complexity make them a popular choice for home automation, transportation, traffic management, and Industrial Control Systems (ICS). These advantages come with a heightened risk of unauthorized data access and modifications to ZigBee networks. Some of these risks include rogue devices, replay attacks, network key sniffing, spoofing, and denial of service attacks [2–4]. Digital communication systems typically implement security mechanisms using bit-level credentials and validity checks to guard against such attacks. This research focuses on leveraging unique properties within

Physical Layer (PHY) waveform transmissions as an additional layer of security to complement traditional security measures.

1.2 Technical Motivation

There has been an extensive amount of research conducted at the Air Force Institute of Technology (AFIT) investigating techniques to augment the security of numerous communication systems by exploiting features found in the PHY [5–19]. Part of AFIT’s research portfolio includes using Radio Frequency-Distinct Native Attribute (RF-DNA) fingerprinting as an effective a method to augment the security of ZigBee based networks. RF-DNA Fingerprinting generates human-like “fingerprints” from a device’s PHY waveform to attain a unique one-to-one association between a fingerprint and a device. This makes RF-DNA difficult to mimic and serves as an effective method to classify various devices. While RF-DNA classification has shown to be effective, a majority of the previous experiments have been implemented in the form of MATLAB based post-processing This research is an initial step towards the ultimate goal of demonstrating an autonomous low-cost, compact, stand-alone Radio Frequency (RF) air monitor. This will be accomplished by implementing an air monitor composed of an Software-Defined Radio(s) (SDR) loaded with a trained Multiple Discriminant Analysis/Maximum Likelihood (MDA/ML) model; assessing the air monitor’s NRT fingerprinting and discrimination performance; and comparing the air monitor’s results to MATLAB results.

1.3 Document Organization

The remainder of this document is organized as follows:

- Chapter II - Background: Provides a brief overview of the ZigBee protocol, the modulation methods involved to process ZigBee transmissions, RF-DNA finger-

printing, the MDA/ML process, and the hardware and software utilized throughout this research.

- Chapter III - Methodology: Provides the methodology used for experimental signal collection, the SDR design, the two fingerprinting generation methods, and the MDA/ML process and data generation.
- Chapter IV - Analysis of Results: Provides results and analysis of the air monitor's performance during high disk usage; the air monitor's performance loaded with trained MDA/ML data; the estimative and actual MDA/ML models; and NRT device discrimination results.
- Chapter V - Conclusion: Provides a summary and conclusions based on research results and recommendations for future research.

II. Background

This chapter provides background on the methods used in this research supporting Chapter III and the interpretation of the results generated in Chapter IV. Section 2.1 provides a brief overview of the ZigBee protocol defined by the Institute of Electrical and Electronics Engineers (IEEE) 802.15.4 standard for Wireless Personal Area Network(s) (WPAN) [20]. Section's 2.2 and 2.3 provide a general overview of Offset-Quadrature Phase Shift Keying (O-QPSK) and Minimum Shift Keying (MSK) modulation. Section 2.4 goes over Radio Frequency-Distinct Native Attribute (RF-DNA) Fingerprinting which consists of extracting statistical features from specified Region(s) of Interest (ROI) in the time-domain. Section 2.5 covers model development and device classification using a Multiple Discriminant Analysis/Maximum Likelihood (MDA/ML) process. Section 2.6 describes equipment used to isolate Radio Frequency (RF) signals. Section 2.7 describes KillerBee, a security research toolkit specifically designed to explore, evaluate, and exploit IEEE 802.15.4 based networks. Section 2.8 covers Wireshark, a network protocol analyzer to inspect packets [21]. Section 2.9 briefly covers the Cyclic Redundancy Check (CRC)-16 error-checking used in this research. This is followed by a very brief description of a C++ based linear algebra library in Section 2.10. Section 2.11 goes over the Software-Defined Radio(s) (SDRs) used in this research. Chapter II wraps up with Section 2.12 which describes GNU Radio, an open-source SDR framework [22].

2.1 ZigBee

ZigBee is an IEEE 802.15.4 based standard for very low-cost, low-data rate, low-power WPAN [20, 23]. ZigBee devices are utilized for consumer electronics, home and building automation, industrial controls, PC peripherals, and medical sensors in both

military and civilian applications [1, 20, 23]. The IEEE 802.15.4 Physical Layer (PHY) has several modulation standards for various frequencies, but this research is focused on O-QPSK modulated frequency bands which will be explained in Section 2.2 [20, 23].

The IEEE 802.15.4 O-QPSK PHY supports three frequency bands: a 16-channel 2450.0 MHz band; a 10-channel 915.0 MHz band; and a 1-channel 868.0 MHz band. All the previously mentioned IEEE 802.15.4 bands utilize Direct Sequence Spread Spectrum (DSSS) Modulator/Demodulators (MODEMs) [1, 23]. The 2450 MHz band was utilized to conduct all of the experiments for this research. The 16 channels for the 2450 MHz band have a center frequency that can vary between $2405.0 \text{ MHz} \leq f_{rng} \leq 2480.0 \text{ MHz}$ [1, 23]. The standard specifies the center frequency f_C for each of the 16 channels as [1]:

$$f_C = [2405 + 5(k - 11)] \text{ MHz, for } k = 11, \dots, 26 \quad (2.1)$$

Each channel is separated by $\Delta_{Ch} = 5.0 \text{ MHz}$ with an instantaneous bandwidth of $f_{BW} = 2.0 \text{ MHz}$ and intra-channel gap-bands of $f_{gap} = 3.0 \text{ MHz}$ [1, 23]. The center frequencies and bandwidth utilization are illustrated in Figure 2.1.

ZigBee incorporates DSSS by mapping 4-bit segments to one of sixteen Data Symbol(s) (DS) [23]. Each DS is then mapped to a nearly orthogonal 32-bit sequence as shown in Table 2.1 [1, 23]. The even DS bits are mapped to the In-Phase (I) component of the O-QPSK signal and the odd DS bits are mapped to the offset Quadrature-Phase (Q) component of the O-QPSK signal [1, 23]. Previous research conducted by

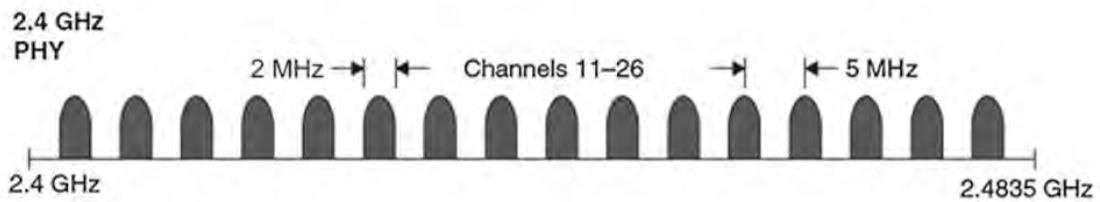


Figure 2.1. Frequencies and Channel Spread for ZigBee in the 2450.0 MHz Band [1]

Dr. Thomas Schmidt demodulated IEEE 802.15.4 signals using a non-coherent receiver employing MSK [24]. Section 2.3 has more details on about how to implement MSK as a special case of O-QPSK [24, 25]. Demodulating IEEE 802.15.4 with a non-coherent receiver with MSK will change two components. First, the DSSS mapping sequence will be different than the O-QPSK as shown on Table 2.2. Second, the first demodulated DSSS chip is always set to a "don't care" value because the I bit will be unknown and both In-Phase and Quadrature-Phase (I/Q) chips are needed to determine the proper value of the first chip [24].

ZigBee devices operating in the 2450.0 MHz band have a data rate (R_b), DS symbol rate (R_{DS}), chip-rate (R_c), O-QPSK symbol rate (R_{O-QPSK}) as follows:

$$\begin{aligned}
 R_b &= 250.0 \text{ kBits/sec} \\
 R_{DS} &= 62.5 \text{ kSym/sec} \\
 R_c &= 2.0 \text{ Mc/sec} \\
 R_{O-QPSK \text{ Sym}} &= 1.0 \text{ MS/sec}
 \end{aligned} \tag{2.2}$$

Section 2.2 provides additional information about O-QPSK modulation scheme. Figure 2.2 provides an overall illustration of the ZigBee DSSS process along with each symbols corresponding 32-Bit Unsigned Integer (uInt32) value [1, 23].

Per the IEEE 802.15.4 standard, the O-QPSK PHY Protocol Data Unit (PPDU) consists of a Synchronization Header (SHR) that is used to synchronize to a data stream; a PHY Header (PHR) containing the frame length; and a variable length PHY Service Data Unit (PSDU) that contains the remaining data and sublayer [1, 9, 23]. The SHR is composed of a preamble sequence consisting of 4 bytes of 0x00 and a Start-of-Frame Delimiter (SFD) byte of 0x7A (or b1110 0101) marking the end of the SHR [23]. This

Table 2.1. ZigBee Symbol-to-Chip Mapping for the O-QPSK 2450.0 MHz Band [20, 24]

Symbol	Chip sequence (C0, C1, C2, ... , C31)	uInt32 value
0	11011001110000110101001000101110	3653456430
1	11101101100111000011010100100010	3986437410
2	00101110110110011100001101010010	786023250
3	00100010111011011001110000110101	585997365
4	01010010001011101101100111000011	1378802115
5	00110101001000101110110110011100	891481500
6	11000011010100100010111011011001	3276943065
7	10011100001101010010001011101101	2620728045
8	10001100100101100000011101111011	2358642555
9	10111000110010010110000001110111	3100205175
10	01111011100011001001011000000111	2072811015
11	01110111101110001100100101100000	2008598880
12	00000111011110111000110010010110	125537430
13	01100000011101111011100011001001	1618458825
14	10010110000001110111101110001100	2517072780
15	11001001011000000111011110111000	3378542520

Table 2.2. ZigBee Symbol-to-Chip Mapping for the MSK 2450.0 MHz Band [24]

Symbol	Chip sequence (C0, C1, C2, ... , C31)	uInt32 value
0	x1100000011101111010111001101100	1618456172
1	x1001110000001110111101011100110	1309113062
2	x1101100111000000111011110101110	1826650030
3	x1100110110011100000011101111010	1724778362
4	x0101110011011001110000001110111	778887287
5	x1111010111001101100111000000111	2061946375
6	x1110111101011100110110011100000	2007919840
7	x0000111011110101110011011001110	125494990
8	x0011111100010000101000110010011	529027475
9	x0110001111110001000010100011001	838370585
10	x0010011000111111000100001010001	320833617
11	x0011001001100011111100010000101	422705285
12	x1010001100100110001111110001000	1368596360
13	x0000101000110010011000111111000	85537272
14	x0001000010100011001001100011111	139563807
15	x1111000100001010001100100110001	2021988657

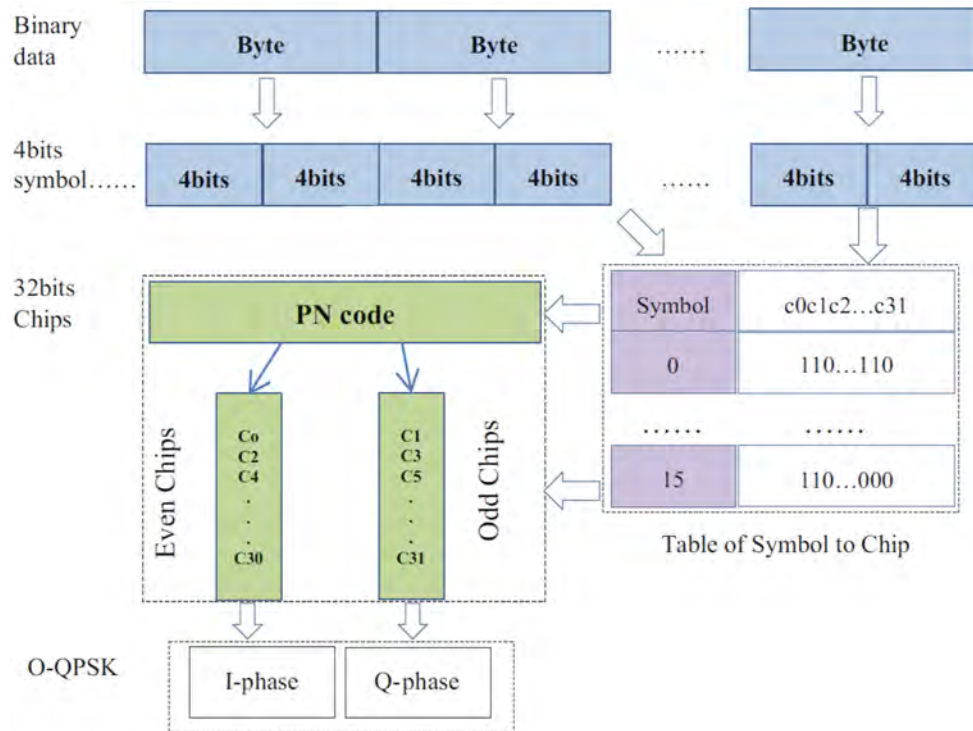


Figure 2.2. Flowchart of ZigBee's DSSS Implementation [1]

research was accomplished by segregating devices based on the statistical data gathered from the waveform containing the SHR of each device. Sections 2.4 and 2.5 cover how each device was identified using statistical data. Figure 2.3 shows a complete IEEE 802.15.4 PPDU packet.

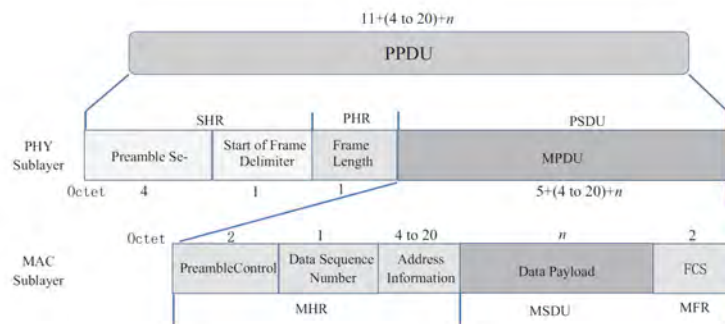


Figure 2.3. PPDU Frame Format [1, 9, 23]

2.2 O-QPSK Modulation

Quadrature Phase Shift Keying (QPSK) modulation consists of two independent Binary Phase Shift Keying (BPSK) signals: an I signal and a Q signal [25]. QPSK divides a binary input data stream $d_{Data}(t) = d_0, d_1, d_2, \dots$ to separate channels representing 1 or 0 as bipolar pulses where $d_{Data} = \pm 1$ [25]. The even data bits are inserted onto the I data stream and the odd bits do the same on the Q data stream as shown on (2.3) and (2.4) [25].

$$d_I(t) = d_0, d_2, d_4, \dots \text{ (even bits)} \quad (2.3)$$

$$d_Q(t) = d_1, d_3, d_5, \dots \text{ (odd bits)} \quad (2.4)$$

The $d_I(t)$ and $d_Q(t)$ data streams are modulated onto the I and Q portions of the carrier wave as follows [25]:

$$s(t) = \frac{1}{\sqrt{2}} d_I(t) \cos\left(2\pi f_0 t + \frac{\pi}{4}\right) + \frac{1}{\sqrt{2}} d_Q(t) \sin\left(2\pi f_0 t + \frac{\pi}{4}\right), \quad (2.5)$$

where f_0 represents the carrier frequency in Hz.

Since the I and Q QPSK data streams are orthogonal to each other, the two data streams can be demodulated separately [25]. Using the trigonometric identities, QPSK can be written in the following form:

$$s(t) = \cos(2\pi f_0 t + \theta(t)), \quad \text{where } \theta = \left[0, \frac{\pi}{2}, \frac{3\pi}{2}, \pi\right]. \quad (2.6)$$

O-QPSK is equivalent to QPSK waveforms in that both modulation schemes share the same constellation symbols and can also be represented by (2.5) [25]. What sets O-QPSK and QPSK apart is that QPSK can transition from one symbol to any other symbol as shown in Figure 2.5, while O-QPSK can only make a transition to adjacent symbols (i.e. $\Delta\theta \in [-\pi/2, 0, \pi/2, \pi]$) as shown on Figure 2.6 [25]. This is accomplished

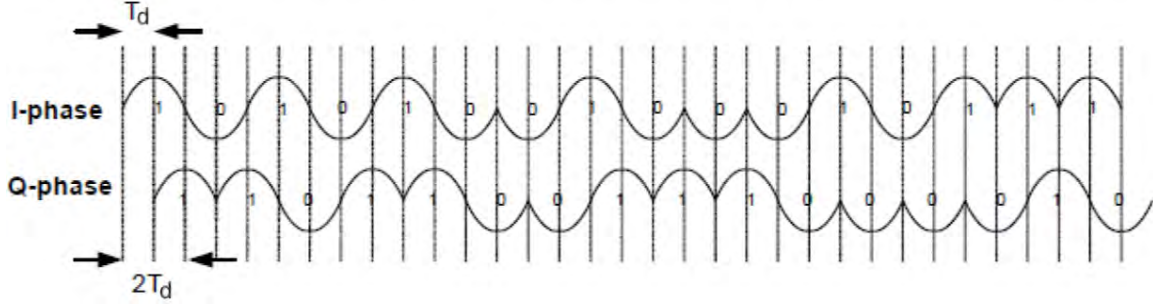


Figure 2.4. O-QPSK I/Q Waveform with Time Offset T_d [23, 25]

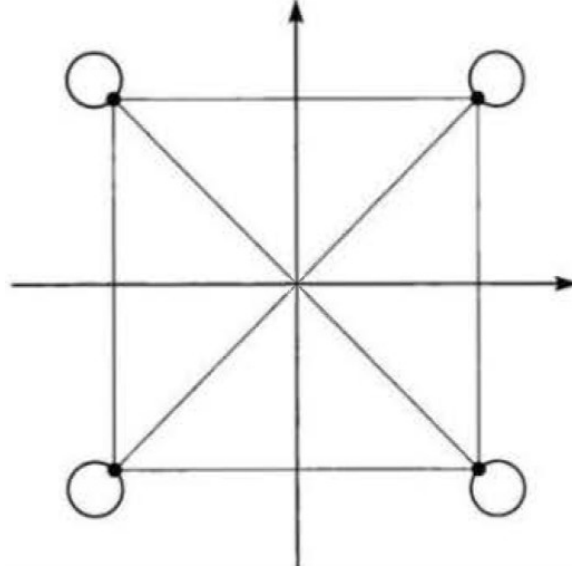


Figure 2.5. QPSK Phase Transition Diagram [26]

by adding a time offset T_d that is equal to half of the symbol period T_{O-QPSK} . The I and Q data streams are represented on Figure 2.4.

2.3 MSK Modulation

MSK is a form of Continuous Phase Modulation (CPM) that can be viewed as a special case of O-QPSK with sinusoidal symbol weighting [25]. MSK data encoding using sinusoidal symbol weighting can be represented as

$$s(t) = d_I(t) \cos\left(\frac{\pi t}{2T}\right) \cos(2\pi f_0 t) + d_Q(t) \sin\left(\frac{\pi t}{2T}\right) \sin(2\pi f_0 t), \quad (2.7)$$

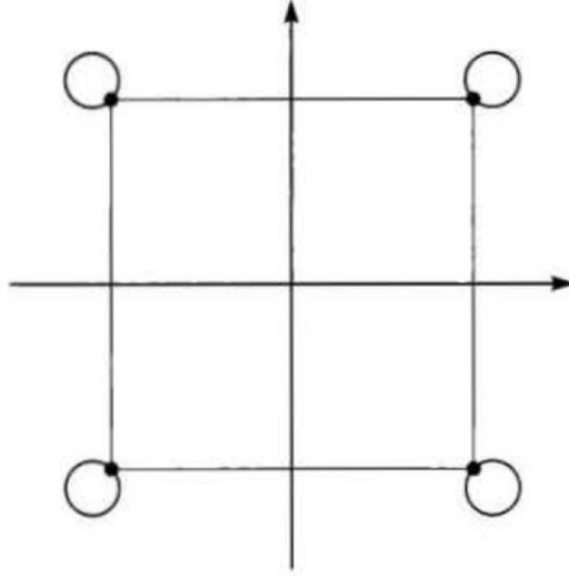


Figure 2.6. O-QPSK Phase Transition Diagram [26]

with $d_I(t)$ and $d_Q(t)$ using the same data stream as (2.3) and (2.4).

2.4 RF-DNA Fingerprinting

Fingerprinting is a method of creating a mathematical profile of a device based on collected emissions. Factors that influence each device's fingerprint include operating temperature, device age, tolerances, and variations in manufacturing [5, 8–10, 27]. RF-DNA fingerprints can be generated from Time Domain (TD) or Spectral Domain (SD) responses. This research effort focused on TD fingerprints and do not address SD responses. The fingerprints are generated using the instantaneous amplitude (a), instantaneous phase (ϕ), and instantaneous frequency (f) responses of a specified ROI. The ROI is broken down into N_R subregions where each instantaneous feature is assessed. Individual instantaneous features are calculated within an ROI subregion N_R containing N_s complex samples of equal sample length as

$$N_s = \lfloor \text{length}(\text{ROI}) / N_R \rfloor. \quad (2.8)$$

With the N_s samples in each subregion N_R , each instantaneous feature is calculated using the Real ($\Re$) and Imaginary ($\Im$) components from $\mathbf{x}[n]$ as follows [9]:

Instantaneous Amplitude (a)

$$\mathbf{a}[n] = \sqrt{\Re(\mathbf{x}[n])^2 + \Im(\mathbf{x}[n])^2}, \quad n = \{0, 1, 2, \dots, N_s - 1\}, \quad (2.9)$$

Instantaneous Phase (ϕ)

$$\phi[n] = \tan^{-1} \left(\frac{\Im(\mathbf{x}[n])}{\Re(\mathbf{x}[n])} \right), \quad \Im(\mathbf{x}[n]) \neq 0, \quad n = \{1, 2, \dots, N_s - 1\}, \quad (2.10)$$

Instantaneous Frequency (f)

$$\mathbf{f}[n] = \frac{d\phi[n]}{d[n]}, \quad n = \{0, 1, 2, \dots, N_s - 1\}. \quad (2.11)$$

Previous research has shown that an IEEE 802.15.4 SHR waveform serves as an ideal ROI since the waveform within the SHR contains the same first 5 bytes (10 DSSS symbols) for every transmitted burst [9, 16].

The sequences from (2.9) through (2.11) are adjoined subregions from the ROI and summarized using $N_M = 3$ statistical metrics of variance (σ^2), skewness (γ), and kurtosis (κ) for each N_R subregion that are calculated from,

$$\mu = \frac{1}{N_s - 1} \sum_{n=0}^{N_s-1} \mathbf{x}[n]. \quad (2.12)$$

$$\sigma^2 = \frac{1}{N_s - 1} \sum_{n=0}^{N_s-1} (\mathbf{x}[n] - \mu)^2, \quad (2.13)$$

$$\gamma = \frac{1}{(N_s - 1)\sigma^3} \sum_{n=0}^{N_s-1} (\mathbf{x}[n] - \mu)^3, \quad (2.14)$$

$$\kappa = \frac{1}{(N_s - 1)\sigma^4} \sum_{n=0}^{N_s-1} (\mathbf{x}[n] - \mu)^4, \quad (2.15)$$

and used to form the ROI subregion vector as

$$\mathbf{F}_{R_i} = [\sigma^2, \gamma, \kappa]_{1 \times 3}. \quad (2.16)$$

The full-dimensional time domain fingerprint N_{TD} is composed of N_R subregions, N_F instantaneous features per subregion, and N_M statistical metrics per instantaneous TD feature as shown on (2.17) and (2.18). Figure 2.7 is a summarized overview of RF-DNA TD Fingerprinting.

$$\mathbf{F}_{TD} = \left[\mathbf{F}^1 : \mathbf{F}^2 \dots : \mathbf{F}^N \right]_{[1 \times N_M \times N_R \times N_F]} = \left[\mathbf{F}^a : \mathbf{F}^\phi : \mathbf{F}^f \right]_{[1 \times N_M \times N_R \times 3]} \quad (2.17)$$

$$\mathbf{N}_F = [1 \times N_M \times N_R \times N_F] \quad (2.18)$$

2.5 MDA/ML

Fingerprints are created specifically to classify and discriminate between devices based on statistical metrics. Multiple Discriminant Analysis (MDA) and Maximum Likelihood (ML) processing effectively classifies and maximizes device discrimination with provided fingerprints. First part of the process begins with MDA reducing input feature dimensionality by projecting N_F input features into a N_C-1 subspace where it is assumed that $N_F \geq N_C$ and $N_C > 2$ in an attempt to maximize between-class dimensional spread and minimize within-class dimensional spread. The between-class (S_b) and within-class (S_w) scatter matrices are computed as,

$$\mathbf{S}_b = \sum_{i=1}^{N_C} \mathbf{P}_i \Sigma_i, \quad (2.19)$$

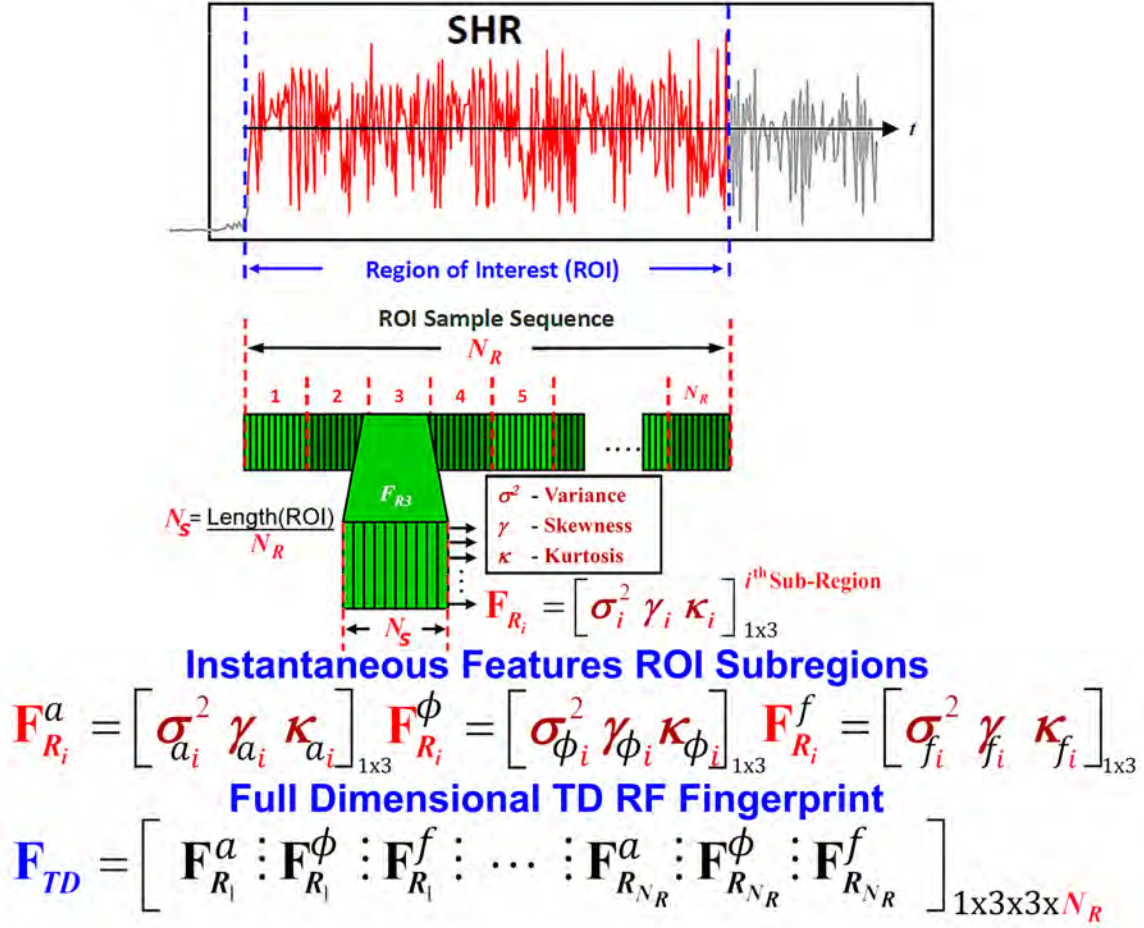


Figure 2.7. Summarized Overview of RF-DNA TD Fingerprinting [14, 27]

$$\mathbf{S}_w = \sum_{i=1}^{N_C} \mathbf{P}_i (\mu_i - \mu_0)(\mu_i - \mu_0)^T, \quad (2.20)$$

with class covariance (Σ_i) and global mean (μ_0) calculated as,

$$\Sigma_i = E \left[(x - \mu_i)(x - \mu_i)^T \right], \quad (2.21)$$

$$\mu_0 = \sum_{i=1}^{N_C} P_i \mu_i, \quad (2.22)$$

where μ_i is the mean and P_i is the prior probability of each N_C class [9]. The generated $\mathbf{F}_{TD}$ RF-DNA fingerprints are projected into the $(N_C - 1)$ -dimensional space in

accordance with

$$\hat{\mathbf{f}} = \mathbf{W}^T \mathbf{F}_{\text{TD}}, \quad (2.23)$$

where $\mathbf{W}$ is the $N_F \times (N_C - 1)$ projection matrix created from the $N_C - 1$ Eigenvectors of $\mathbf{S}_w^{-1} \mathbf{S}_b$ and $\hat{\mathbf{f}}$ is the RF-DNA fingerprint projection into the lower dimensional subspace.

The effectiveness of $\mathbf{W}$ classifying devices entirely depends on how well $\mathbf{S}_b$ distance was maximized and $\mathbf{S}_w$ spread was minimized. Figure 2.8 is an example of two MDA projection matrices, $\mathbf{W}_1$ and $\mathbf{W}_2$, for $N_C = 3$ classes onto a $(N_C - 1)$ -dimensional subspace with $\mathbf{W}_1$ providing the "best" classification [27].

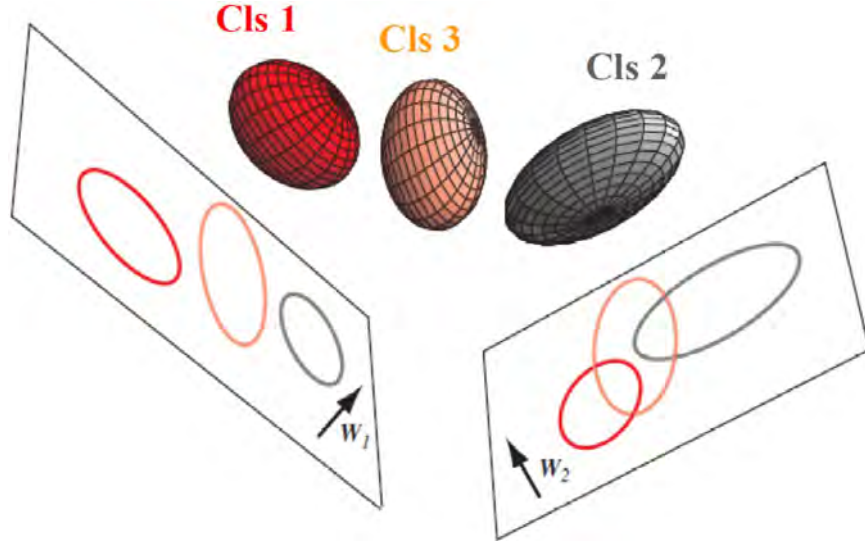


Figure 2.8. Representative MDA Projections using $\mathbf{W}_1$ and $\mathbf{W}_2$ for a $N_C = 3$ problem into a $(N_C - 1)$ -Dimensional Space [27].

ML classification of $\hat{\mathbf{f}}$ is implemented by performing an estimation based on Bayesian posterior probability that the conditional posterior probability $P(c_i | \hat{\mathbf{f}})$ that $\hat{\mathbf{f}}$ belongs to class c_i where $i = 1, 2, \dots, N_C$ is given as [8, 27, 28]:

$$P(c_i | \hat{\mathbf{f}}) = \frac{P(\hat{\mathbf{f}} | c_i) P(c_i)}{P(\hat{\mathbf{f}})} \quad (2.24)$$

The final classification decision is made by designating the calculated $\hat{\mathbf{f}}$ to the N_C returning the highest probable measure of similarity.

2.6 Ramsey STE4400

The Ramsey STE4400 is a portable benchtop RF Shielded test enclosure that enables testing and analysis in a tightly controlled test environment. The STE4400 provides 90 dB attenuation for signals in the 2 GHz band operating outside of the test enclosure [29].

2.7 KillerBee: ZigBee Attack Platform

KillerBee was developed by River Loop Security as a framework and toolset for attacking IEEE 802.15.4 based networks to include ZigBee [4]. KillerBee enables the AT-MEL RZUSBsticks devices to transmit arbitrary ZigBee packets and/or beacon requests even when they are not operating in a network. The transmitted data can be compared against the received collections for statistical analysis and validation of results. The following KillerBee tools were leveraged for generating data [4]:

- `zbreplay` - Implements a replay attack from a specified daintree file or libpcap packet capture. acknowledgement (ACK) frames are not retransmitted.
- `zbconvert` - Convert a packet capture from libpcap to daintree Sensor Network Analyzer (SNA) format, or vice-versa.

2.8 Wireshark

Wireshark is an open-source network protocol analyzer that provides users the ability to exhaustively inspect network data in a live or off-line environment [21]. Wireshark was used as a verification tool since the software suite already has support for

both ZigBee and IEEE 802.15.4 protocols. The network traffic for a ZigBee device from the Control4[®] product line was stored in a libpcap file for use as a baseline. These network traffic was inspected with Wireshark to assess normal ZigBee traffic.

2.9 CRC-16

CRC is an error-detection code and a sub-class of linear block codes used to verify the integrity of the data [30]. The derivation of the formula is outside the scope of this thesis, but CRC works by performing polynomial long division on a set of data $d(x)$ taken as the dividend and a generator polynomial $G(x)$ as the divisor [30]. The long divisions resultant remainder $R(x)$ is the parity bit used to determine if the data is valid. IEEE 802.15.4 PSDU packets contain and make use of a 16-bit International Telecommunication Union - Telecommunication Standardization Sector (ITU-T) CRC generator polynomial $G_{16}(x)$. The generator polynomial $G_{16}(x)$ is defined as

$$G_{16}(x) = x^{16} + x^{12} + x^5 + 1, \quad (2.25)$$

where an $R(x)$ with an odd result indicates that there is an error in the message. An CRC-16 error-check was performed on the recovered ZigBee packets in this research for statistical analysis. A complete description of the CRC-16 usage is presented in Chapter IV.

2.10 Eigen

Eigen is an open-source C++ template library for linear algebra to support matrix, vector, numerical solvers, and related operations and algorithms [31]. Eigen was used for matrix multiplication in Near Real-Time (NRT) MDA/ML that will be further expanded upon in Chapter III.

2.11 SDR

An SDR is a radio device in which some or all of the functions are software defined [32]. SDRs are extremely flexible devices that allow users to configure modulation, bandwidth, and sampling rates on the fly as opposed to purely analog devices that have a fixed configuration. An Ettus B205mini-i was used to collect ZigBee transmissions for this research. The B205mini-i has a tuning range of $70.0 \text{ MHz} \leq f_{rng} \leq 6.0 \text{ GHz}$, an instantaneous bandwidth of $f_{BW} \leq 56.0 \text{ MHz}$, and an adjustable sample rate up to $R_{samp} = 61.44 \text{ MS/sec}$ with a resolution of 12-bits [33].

2.12 GNU Radio

GNU Radio is a free open-source software development kit that leverages SDRs through the use of signal processing blocks in a live or simulated environment [22]. Users have the ability to rapidly create and reconfigure a signal processing flow graph using a built-in tool called *gnuradio-companion*. *gnuradio-companion* is an Graphical User Interface (GUI) providing the ability to quickly drag-and-drop signal blocks on a flow graph and connect them together. This tool implements signal processing blocks that rivals in performance with commercial applications such as MathWorks Simulink or NI-LabVIEW. The *gnuradio-companion* flow graph generates a python file that can be further modified to further customize the generated GUI or include additional python libraries that aren't included within *gnuradio-companion*.

GNU Radio has a wide variety of common Digital Signal Processing (DSP) blocks that implements common operations encountered in digital communication systems. However, this research required a custom signal processing to generate RF-DNA fingerprints in NRT. GNU Radio's *gr_modtool* script facilitates the creation of custom block by generating templates for the required files. *gr_modtool* enables a user to create Out-of-Tree (OOT) modules that house new blocks generated in Python or C++.

The performance requirements to properly fingerprint and discriminate ZigBee bursts in NRT required the development of a C++ block.

A complete IEEE 802.15.4 SHR waveform is required to be captured accurately and processed almost immediately for NRT device discrimination. An existing OOT module developed by Dr. Bastian Bloessl was modified to capture valid SHR transmissions during DSSS demodulation [34, 35]. Another block was developed to perform NRT RF-DNA fingerprinting and MDA/ML discrimination. Chapter III elaborates the details involved in performing NRT RF-DNA fingerprinting and MDA/ML discrimination.

III. Methodology

This chapter provides the methodology used to acquire results posted in Chapter IV. Each section is presented in successive order as shown on Figure 3.1. Section 3.1 explains the GNU Radio Software-Defined Radio(s) (SDR) air monitor's physical and software receiver design. Section 3.2 provides detailed specifics on how the Institute of Electrical and Electronics Engineers (IEEE) 802.15.4 Synchronization Header (SHR) bursts were captured with the blocks described in Section 3.1.2. Section 3.3 describes the fingerprint generation process completed in MATLAB and GNU Radio. Section 3.4 specifies the devices discriminated using Multiple Discriminant Analysis/Maximum Likelihood (MDA/ML) processing and how the trained data model was applied to the blocks in Section 3.1.2.

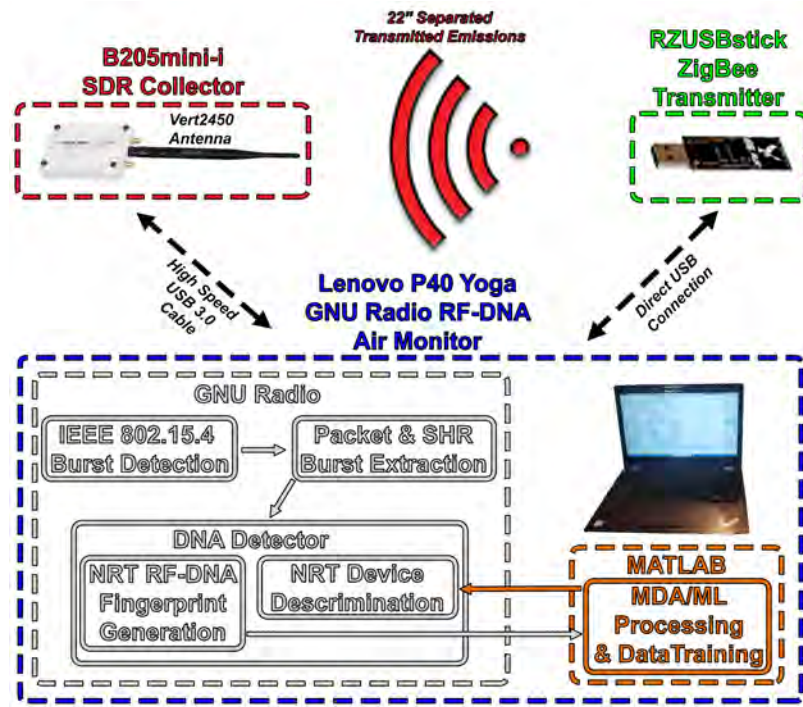


Figure 3.1. Methodology for NRT RF-DNA Fingerprinting and Device Discrimination

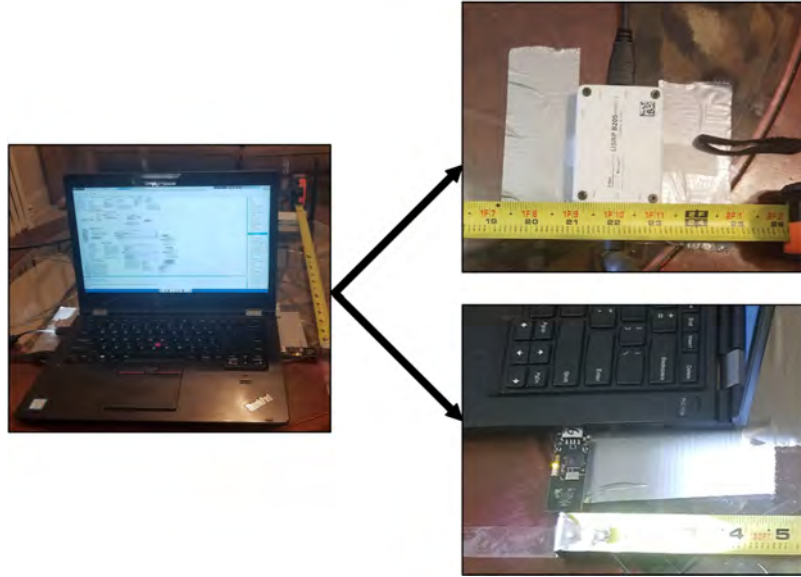
3.1 GNU Radio SDR Receiver Design

The hardware used by the air monitor in this research consists of an Ettus B205mini-i, a Lenovo p40 Thinkpad Yoga laptop, and various RZUSBsticks that were switched between collection and discrimination trials as illustrated on Figure 3.2. The Lenovo p40 is equipped with an Intel[®] Core[™] i7-660 Processor, 16 GB of Low Voltage Double Data Rate Type 3 Synchronous Dynamic Random-Access Memory (DDR3L), a 1 TB Samsung 850 Pro Solid State Drive (SSD), and an NVIDIA[®] Quadro[®] M500M Graphics Processing Unit (GPU) with 2 GB of Double Data Rate Type 3 Synchronous Dynamic Random-Access Memory (DDR3). The B205mini-i was connected to the Lenovo p40 using a 4 feet Universal Serial Bus (USB) 3.0 cable capable of sustaining data rates of up to 625 MB/sec [36]. This setup enables maximum performance for the B205mini-i's since this device data rate is limited to 184.3 MB/sec as listed on Section 2.11. Each RZUSBstick had a direct connection to the Lenovo p40's USB port positioned closest to the B205mini-i. The B205mini-i and each RZUSBstick were separated by approximately 22 inches for every trial. To ensure a consistent trial setting, the locations for the Lenovo p40 and B205mini-i were marked in the test bench, and the B205mini-i was secured in place using masking tape. Table 3.1 contains the manufacturing label information for each RZUSBstick.

Ubuntu 18.04.1 LTS was the Operating System (OS) used to develop the GNU Radio air monitor and operate the tools discussed in Chapter II. GNU Radio was compiled with build version *3.14.13.4* and the B205mini-i's drivers were compiled from *release_003_011_000_1*. Refer to Ettus and GNU Radio's websites for detailed software installation instructions and the required dependencies needed for everything to work properly [22, 33]. The Eigen C++ libraries were installed alongside GNU Radio to implement linear algebra algorithms required to generate RF-DNA fingerprints and perform device discrimination. The GNU Radio modules *gr-ieee802.15.4* and *gr-*

Table 3.1. RZUSBstick's MAC Address, AT86RF230 Transceiver Marks, Device Date Code, and SN

Device	MAC ID	Mark 1	Mark 2	Date Code	Device SN
1	A0:F6:9F:E7	1442 PH	1R8338-7	2015.05.26	0200004417
2	A0:01:43:70	0923 PH	8P0772	2010.06.01	0200002338
3	A0:01:5D:34	0936 PH	9P0187-2	2015.05.26	0200004387
4	A0:F6:A0:68	1442 PH	1R8338-7	2010.05.31	0200002058
5	A0:F6:A0:4E	1442 PH	1R8338-7	2015.05.26	0200004528
6	A0:F6:9F:FF	1442 PH	1R8338-7	2015.05.26	0200004485
7	A0:F6:A0:0C	1442 PH	1R8338-7	2015.05.26	0200004488
8	A0:F6:A0:04	1442 PH	1R8338-7	2015.05.26	0200004508
9	A0:F6:9F:EA	1442 PH	1R8338-7	2015.05.26	0200004377
10	A0:F6:9F:E0	1442 PH	1R8338-7	2015.05.26	0200004416

**Figure 3.2. Air Monitor Test Configuration**

foo contain the necessary flow graphs, blocks, and source code necessary to develop the air monitor. Both modules are publicly available under the GNU General Public License (GPL) agreement from Dr. Bastian Bloessl's GitHub repository [34, 37].

An *IEEE 802.15.4 Offset-Quadrature Phase Shift Keying (O-QPSK) Physical Layer (PHY)* transceiver block within the *gr-ieee802.15.4* module has been shown to effectively capture and process IEEE 802.15.4 packets [24, 34, 35]. Several features were extracted from the *IEEE 802.15.4 O-QPSK PHY* transceiver block's source code to ac-

curately detect and capture SHR bursts.

Additional GNU Radio blocks were added to the *gr-ieee802.15.4* module to facilitate access to the existing private functions within the *gr-ieee802.15.4* module. Section 3.1.1 explains the *IEEE 802.15.4 O-QPSK PHY* transceiver block illustrated on Figure 3.3. Figure 3.5 illustrates which sub-blocks were extracted and modified to develop the air-monitor. Section 3.1.2 provides a comprehensive overview of the RF-DNA air monitor block components.

3.1.1 *IEEE 802.15.4 PHY* Transceiver Block.

The original *IEEE 802.15.4 PHY* transceiver blocks Python source code was generated in *gnuradio-companion* with the *ieee802_15_4OQPSK_PHY.grc* flow graph file. Figure 3.4 contains the transceiver blocks which receive and transmit IEEE 802.15.4 packets as described in Sections 2.1 and 2.2. The receive blocks and its dependencies were modified to perform NRT fingerprinting and discrimination. The transceiver's receive blocks were implemented using C++ classes available from the base GNU Radio source files or the *gr-ieee802.15.4* module. Further inspection revealed that the Packet Sink block performed the Direct Sequence Spread Spectrum (DSSS) demodulation and passed IEEE 802.15.4 PHY Service Data Unit (PSDU) packets created within the block to the *rxout* asynchronous message port. Each receive block displayed in Figure 3.4 was used in creating the *RFDNA_Prototype_v1.0.grc* flow graph file for the RF-DNA air monitor.

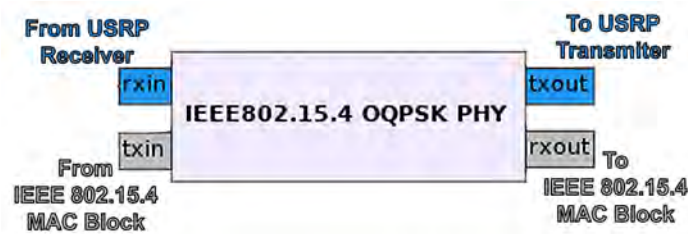


Figure 3.3. *IEEE O-QPSK 802.15.4* Transceiver Block

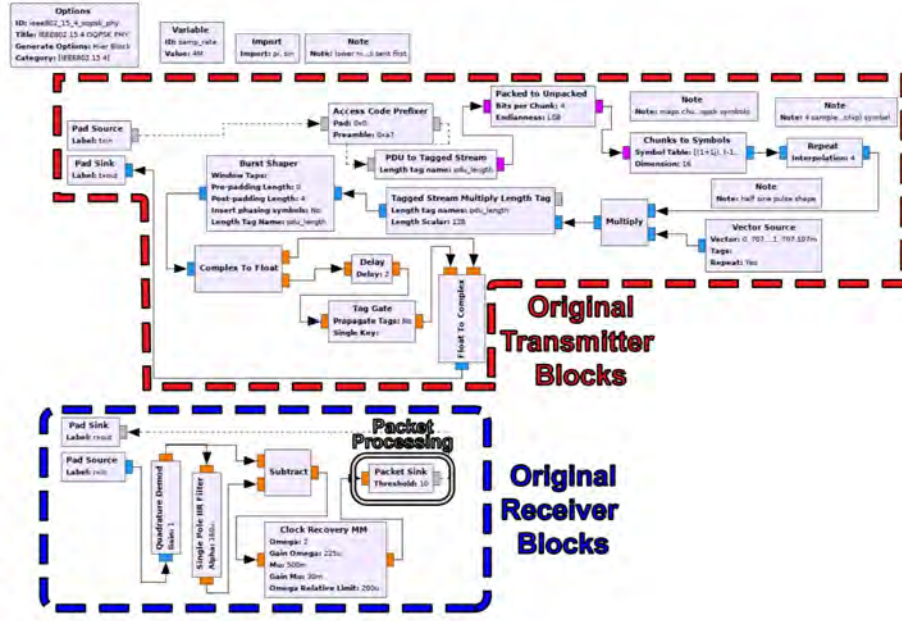


Figure 3.4. IEEE 802.15.4 PHY Transceiver Block Flow Graph

3.1.2 RF-DNA Air Monitor.

The RF-DNA air monitor makes use of the previous blocks extracted from the *IEEE 802.15.4 PHY* transceiver block with the *preamble_sink* block used in place of the *Packet Sink* block. The *preamble_sink* block leveraged *Packet Sink*'s source code and added several features that will be expanded upon in Section 3.1.2.4. The overall functionality of the RF-DNA air monitor can be broken down into 1) Universal Software Radio Peripheral(s) (USRP) and Receive (RX) interface, 2) demodulation and detection, 3) sample delay, 4) data visualization, and 5) data storage. Each functional section of the RF-DNA air monitor's flow graph shown on Figure 3.5 is covered sequentially from Section 3.1.2.1 to Section 3.1.2.6.

3.1.2.1 USRP RX Interface.

The USRP RX interface specifications on Figure 3.6 were kept as close as possible to the examples provided in the *gr-ieee802.15.4* module. As such, the entire flow graph



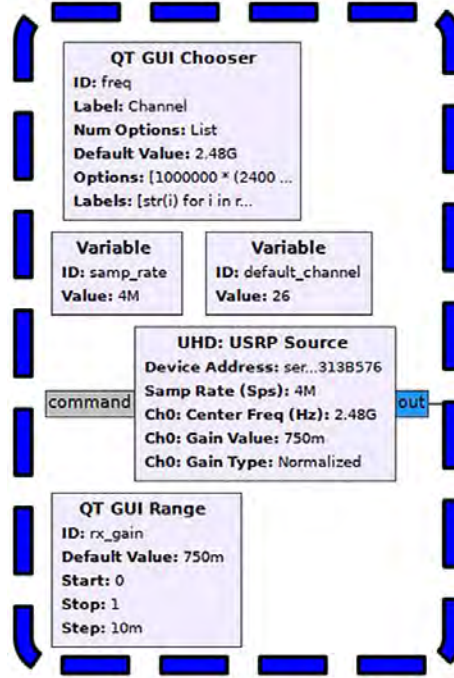


Figure 3.6. USRP RX Interface Blocks

was set to a sample rate $R_{\text{Samp}} = 4.0 \text{ MSamp/sec}$ as a global variable with *samp_rate*. The *QT Graphical User Interface (GUI) Chooser freq* block provides users the ability to change the B205mini-i's center frequency f_C at runtime with a drop-down list of channels between $f_{Ch} = 11$ to 26 based on (2.1). Each channel between $2405.0 \text{ MHz} \leq W_{\text{rng}} \leq 2480.0 \text{ MHz}$ was evaluated to ensure that the air monitor operated as intended, but channel $f_{Ch} = 26$ ($f_C = 2.48 \text{ GHz}$) was the default runtime value passed to *freq* throughout this research. The *QT GUI Range block* provides a slider for the users to modify the normalized receive gain variable *rx_gain*. The *rx_gain* is set to a default normalized receiver gain value $G_{RX\text{Norm}} = 0.75$ that's adjustable in 0.1 increments between $G_{RX\text{Norm}} = 0.0$ to 1.0 during runtime. The *UHD: USRP_Source* block controls and interacts directly with the B205mini-i. The the serial number of the target device must be entered for the device to start. The B205mini-i in this research was configured using the *freq*, *samp_rate*, and *rx_gain* block values on a single RX channel with a normalized gain. The maximum output buffer size of the *UHD: USRP_Source* block was increased

to an arbitrary value of 10 GSamp in the blocks *Advanced* tab to acquire as much buffer space GNU Radio would allow. This was done to help reduce the buffer overflows raw binary RX data, statistical files, fingerprint data, and libpcap files were simultaneously stored during collection trials.

3.1.2.2 Demodulation and Detection.

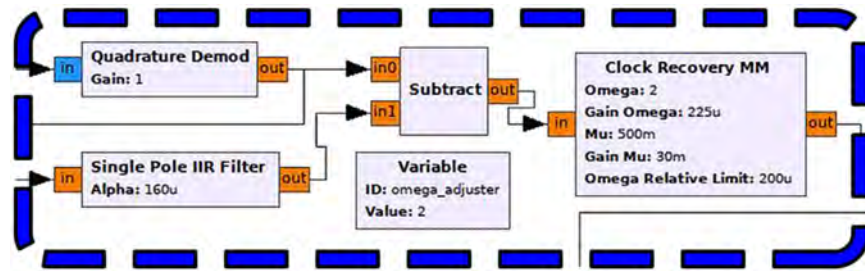


Figure 3.7. Demodulation and Detection Blocks

The demodulation and detection blocks on Figure 3.7 are composed from the *IEEE 802.15.4 O-QPSK PHY* transceiver RX blocks source code from Figure 3.4 and the *omega_adjuster* variable block. The detailed theoretical calculations behind the *Quadrature Demod*, *Single-Pole IIR Filter*, and *Clock Recovery MM* blocks are outside of the scope of this research. The derivations for these block's initial values can be found on GNU Radio's Manual and C++ API website [22]. Functionally, the complex baseband samples received from the *UHD: USRP_Source* block are demodulated by the *Quadrature Demod* block as an Minimum Shift Keying (MSK) transmission. Before the demodulated signal is passed to the *Clock Recovery MM* block to detect RX IEEE 802.15.4 chips, the high frequency components are removed from the collected samples with the *Single Pole infinite impulse response (IIR) Filter* block. The *Clock Recovery MM* block is a Mueller and Müller (M&M) discrete-time error-tracking synchronizer that outputs samples at a rate $R_{MM} = 1 \text{ Sample/Recovered Symbol}$. According to GNURadio's website [22], the *Clock Recovery MM* inputs:

- *Omega* - Sets an initial estimate of samples per symbol.
- *Gain Omega* - Establishes the feedback gain setting for the *Omega* update loop.
- *Mu* - Sets an initial estimate of phase of sample.
- *Gain Mu* - Establishes the feedback gain setting for *Mu* update loop.
- *Omega Relative Limit* - Sets the maximum relative limit

The *Clock Recovery MM* block *Omega* requires adjustment if the sampling rate deviates from $R_{\text{Samp}} = 4 \text{ MSamp/sec}$. The *omega_adjuster* block dynamically adjusts *Omega* using the IEEE 802.15.4 chip period $T_{\text{Chip}} = 0.5 \mu\text{sec/Chip}$ and sampling rate R_{Samp} as

$$\omega = T_{\text{Chip}} \times R_{\text{Samp}}, \quad (3.1)$$

sets *Omega* to $\omega = 2$ samples between each chip as displayed on Figure 3.8. All other values were unchanged from the original *IEEE 802.15.4 O-QPSK PHY* transceiver block design.

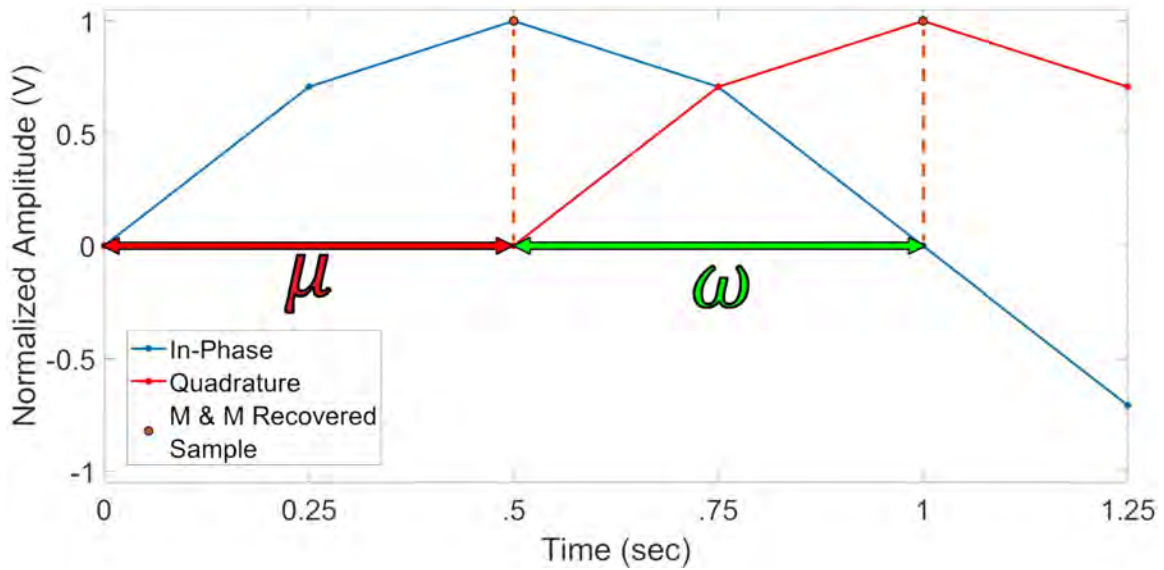


Figure 3.8. GNU Radio Clock Recovery M&M Illustration

3.1.2.3 Sample Delay.

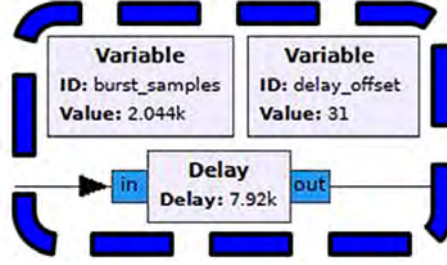


Figure 3.9. Sample Delay Blocks

The complex samples from the *UHD: USRP_Source* block were arriving at the *Packet Sink* block before the detected IEEE 802.15.4 chips could be used to properly capture an SHR sample sequence. The *Delay* block illustrated on Figure 3.9 was added to time shift captured samples by inserting a user specified number of zeroes in front of the received complex sample stream. The size of the delay N_{Delay} was controlled by the *burst_samples* and *delay_offset* variable blocks as

$$N_{\text{Delay}} = \left\lfloor \left(\frac{(\text{burst_samples} \times \text{delay_offset})}{8} \right) \right\rfloor, \quad (3.2)$$

where *burst_samples* is $N_{\text{Samp}} = 2,044$ and *delay_offset* is $N_{\text{Mult}} = 31$. The number of samples N_{Samp} assigned to *burst_samp* was set to match the number of samples the *Packet Sink* block was designed to consume each function call. The *delay_offset* multiplier delays the received complex samples approximately 255 samples at a time contingent on *burst_samp* and (3.2). With a *delay_offset* $N_{\text{Mult}} = 32$ and a sampling period $T_{\text{Samp}} = 250$ nsec based on

$$T_{\text{Samp}} = \frac{1}{R_{\text{Samp}}} \quad (3.3)$$

a delay of $N_{\text{Delay}} = 7,665$ empty samples ($T_{\text{Delay}} = 1,916.25 \mu\text{sec}$) ensured the *UHD: USRP_Source* data stream was received within an appropriate time. The delay multiplier N_{Mult} could vary between different collection configurations, but N_{Mult} was ad-

justed to 32 after inspecting the delay period between several captured SHR samples in MATLAB against IEEE 802.15.4 samples from the B205mini-i.

3.1.2.4 Burst Capture and RF-DNA Processing.

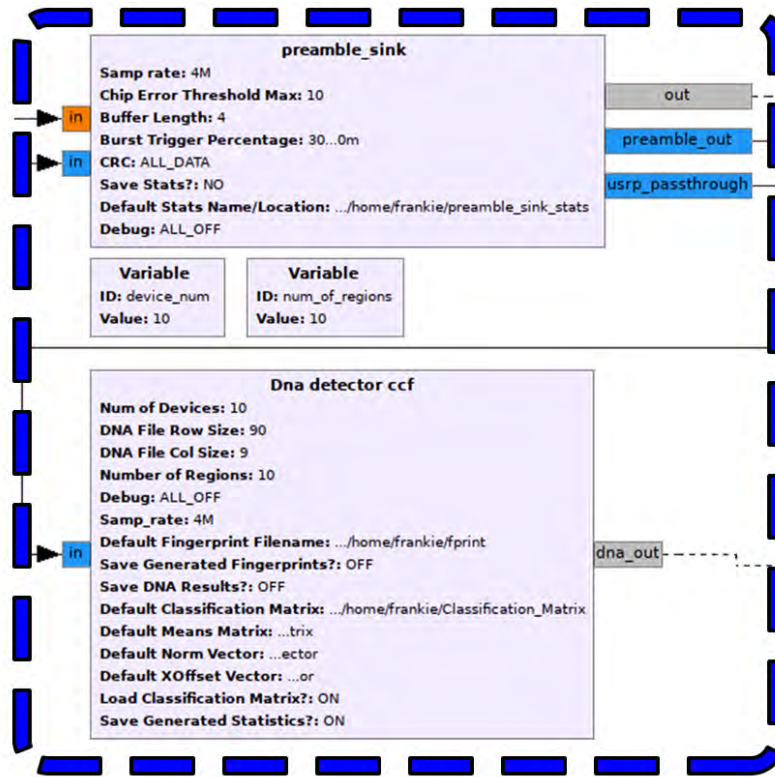


Figure 3.10. Burst Capture and RF-DNA Processing Blocks

The original *Packet Sink* block is a DSSS demodulation state machine that creates and forwards IEEE 802.15.4 PSDU packets to an asynchronous message output. The *preamble_sink* block added an *in* input to receive complex data; an *usrp_passthrough* that forwards received complex data used to detect IEEE 802.15.4 SHR bursts; and several optional data processing inputs. The following options added include:

- *Samp rate* - Automatically adjusts the numbers of samples to capture during burst detection by the input sample rate. *Samp rate* was set $R_{\text{Samp}} = 4\text{MSamp/s}$ by the global *samp_rate* block.

- *Chip Error Threshold Max* - Determines the acceptable number of chip mismatch errors during DSSS demodulation before the state machine is reset. *Chip Error Threshold Max* was set to 10-chips.
- *Buffer Length* - Establishes how many previous bursts are stored within a First-In-First-Out (FIFO) buffer queue. The *Buffer Length* was set to 4-bursts.
- *Burst Trigger Percentage* - Set's the burst detection's trigger threshold as a percentage of a detected and normalized IEEE 802.15.4 burst. *Burst Trigger Percentage* was set to 0.3.
- *Cyclic Redundancy Check (CRC)* - Enables the user to accept any detected burst or just CRC-16 verified bursts (CRC-16 algorithm extracted from *mac* class in the *gr-ieee802-15-4* module). *CRC* was configured to accept any detected bursts.
- *Save Stats* - Enables the user to store statistical information in a text file each time the air-monitor is activated. *Save Stats?* was enabled.
- *Default Stats Name/Location* - Prompts user for the name and location of the statistical file. A timestamp is added to the filename and stored as a text file. *Default Stats Name/Location* was set to store in the root directory formatted with the device identifier, channel number, and block name as "*Emit-ter##_Ch26_preamble_sink_stats*".
- *Debug* - Provides different levels of diagnostic data displayed on the runtime console. *CHIP_ON* displays chip processing information, *PACKET_ON* displays packet information, and *USRP_ON* displays burst information. Any combination of the three options can be enabled. *Debug* was set to *ALL_OFF*.

The complex samples from the B205mini-i had to be captured whenever an SHR binary sequence was detected. Each burst within the *preamble_sink* block was de-

signed to consist of 2,044 complex float samples from the *Delay* block along with 1,022 float samples from the *Clock Recovery MM* block. Both sample streams were not perfectly in sync and the number of received samples occasionally varied, so a buffer of previously collected samples had to be created. A snapshot of the buffered and current B205mini-i data were stored together in a separate buffer when a IEEE 802.15.14 SHR sequence was detected. Once the IEEE 802.15.4 packet is completely filled in the *STATE_HAVE_HEADER* state, the IEEE 802.15.4 SHR waveform stored within the temporary buffer created inside *STATE_SYNC_SEARCH* is detected. This was done by modifying *Packet Sink*'s state machine where:

- *STATE_SYNC_SEARCH* - Continually searches for an a valid SHR sequence until one is found. The current waveform samples are appended to the end of the burst buffer before the entire sample sequence is stored in a temporary buffer for burst detection and the current state is changed to *STATE_HAVE_SYNC*.
- *STATE_HAVE_SYNC* - Searches the PHY Header (PHR) for the packet length and changes the state to *STATE_HAVE_HEADER* if the packet length is valid.
- *STATE_HAVE_HEADER* - Processes and transmits a complete IEEE 802.15.4 PSDU packet. Complex IEEE 801.15.4 SHR waveform samples are extracted during burst detection then loaded to the output signal buffer. The state is reset back to *STATE_SYNC_SEARCH*. Section 3.2 will provide more details on the burst detection process.

The state machine's PSDU packet buffers are cleared and *preamble_sink*'s states are reset any time an invalid DSSS demodulated symbol is detected as shown on Figure 3.11.

The *Dna detector ccf* block performs RF-DNA fingerprinting and device discrimination in NRT on each received SHR burst. The *Dna detector ccf* block settings have

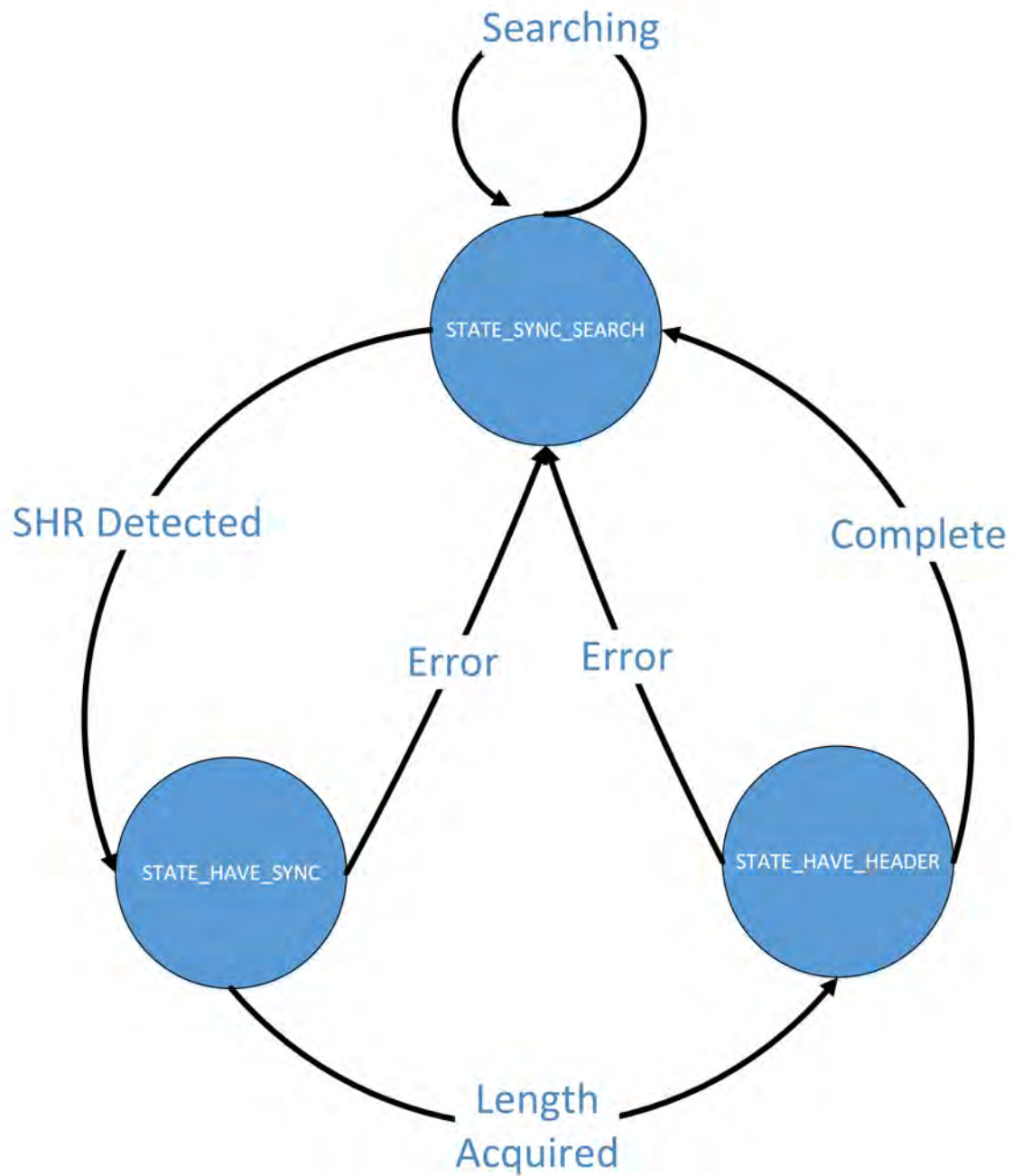


Figure 3.11. Finite State Machine Diagram for *preamble_sink*

different options for the collection and operational trials. The collection trial generated and stored fingerprints, and complex samples to a file. The operational trial generated fingerprints and performs Multiple Discriminant Analysis (MDA) classification in NRT. The data utilized and generated by the *Dna detector ccf* block was located in the root

directory. The *Dna detector ccf*'s block options settings include:

- *Num of Devices* - Number of devices set by the *device_num* block. The *Num of Devices* was set to 10 devices by the *device_num* block.
- *DNA File Row Size* - Row length of the MDA projection matrix $\mathbf{W}$ based on (2.17). *DNA File Row Size* was set to 90.
- *DNA File Col Size* - Column length of the MDA projection matrix $\mathbf{W}$ set by (*device_num* block - 1). *DNA File Col Size* was set to 9 by the *device_num* block.
- *Number of Regions* - Number of regions used for fingerprinting analysis. The *num_of_regions* block was set to 10 regions.
- *Debug* - Provides different levels of diagnostic data displayed on the runtime console like preamble_sink's *Debug* option. *Debug* was set to *ALL_Off*.
- *Samp_rate* - Automatically adjusts the numbers of samples to process during RF-DNA fingerprinting. *Samp_rate* was set $R_{\text{samp}} = 4\text{MSamp/s}$ by the global *samp_rate* block.
- *Default Fingerprint Filename* - Prompts user for the name and location to save the collected fingerprints as a binary data file. The *Default Fingerprint Filename* was set to set to "*Emitter##_Ch26_fprint*".
- *Save Generated Fingerprints* - Enables the user to store RF-DNA fingerprints $\mathbf{F}_{\text{TD}}$ as binary data on a file in NRT. *Save Generated Fingerprints* was enabled during the collection performance trials and disabled for the operational performance trials.
- *Save DNA Results* - Enables the user to store projection results $\hat{\mathbf{f}}$ based on (2.23) as a binary data on a file in NRT. *Save DNA Results* was disabled.

- *Default Classification Matrix* - Prompts user for the location of the MATLAB generated MDA/ML projection matrix $\mathbf{W}$ file.
- *Default Means Matrix* - Prompts user for the location of the MATLAB generated means matrix $\mathbf{W}_\mu$ file.
- *Default Norm Vector* - Prompts user for the location of the MATLAB generated gain normalization vector $\mathbf{F}_{\text{TD Norm}}$ file.
- *Default XOffset Vector* - Prompts user for the location of the MATLAB generated x-offset vector $\mathbf{F}_{\text{TD XOffset}}$ file.
- *Load Classification Matrix* - Enables the user to load the MATLAB generated $\mathbf{W}$, $\mathbf{W}_\mu$, $\mathbf{F}_{\text{TD Norm}}$, and $\mathbf{F}_{\text{TD XOffset}}$ files for NRT device discrimination. *Load Classification Matrix* was disabled during the collection performance trials and disabled for the operational performance trials.
- *Save Generated Statistics* - Enables the user to store two separate timestamped statistical text files. *Save Generated Statistics* was enabled. The "Emiter##_Ch26_fprint_matrix_mult" generated file keeps track of the NRT generated fingerprints $\mathbf{F}_{\text{TD}}$ and detailed discrimination results. The "Emiter##_Ch26_fprint_matrix_mult" file keeps track of statistical information on each device.

Section 3.3.2 provides more detailed information on the NRT fingerprinting process and Section 3.4 provides more information about the MDA/ML support files necessary for NRT device discrimination.

The *Dna detector ccf* block is a state machine as displayed on Figure 3.12 where:

- *SEARCHING* - The *preamble_sink* block generates empty complex floats from the *preamble output* port until an IEEE 802.15.4 SHR is detected. The current state

is changed to a *DETECTED* state once a sample received from the *preamble_sink* block exceeds a threshold value of $1.0e-38$.

- *DETECTED* - Changes the state to *FINGERPRINT* after a sample buffer is filled. The *Samp_rate* input determines how many samples are captured within the buffer.
- *FINGERPRINT* - A fingerprint is created from the buffered SHR samples. MDA analysis is performed on the collected fingerprint using trained data and/or stores fingerprints to create trained data. MDA results are passed to the asynchronous message output and the state is reset back to *SEARCHING*.

3.1.2.5 Data Visualization.

Data was displayed using *QT GUI Histogram Sink* and *QT GUI Time Sink* blocks as shown on Figure 3.13. The *QT Histogram Sink* block is notified by the *Dna detector ccf* block which device was identified. The *device_num* variable block sets the *QT GUI Histogram Sink*'s X-axis. The *QT GUI Time Sink* block displays both the B205mini-i's passed through collections and the detected IEEE 802.15.4 SHR bursts from the *preamble_sink* block. The *preamble_len* and *samp_rate* blocks values are used to set the *QT GUI Time Sink*'s *Number of Points* and *Sample Rate* inputs. The *preamble_len* block calculates the IEEE 802.15.4 SHR sample length as

$$N_{\text{SHR Samples}} = N_{\text{SHR Chips}} \times T_{\text{Chip}} \times R_{\text{Samp}}, \quad (3.4)$$

where *preamble_len* $N_{\text{SHR Samples}} = 640$ samples ($T_{\text{SHR}} = 160 \mu\text{sec}$) when the sampling rate $R_{\text{Samp}} = 4 \text{ MSamp/sec}$, the chip period $T_{\text{Chip}} = 0.5 \mu\text{sec/Chip}$, and $N_{\text{SHR Chips}} = 320$ chips. The RF-DNA air-monitor GUI is composed of the *QT GUI Chooser freq*, *QT GUI*

detector ccf State Diagram.png detector ccf State Diagram.png

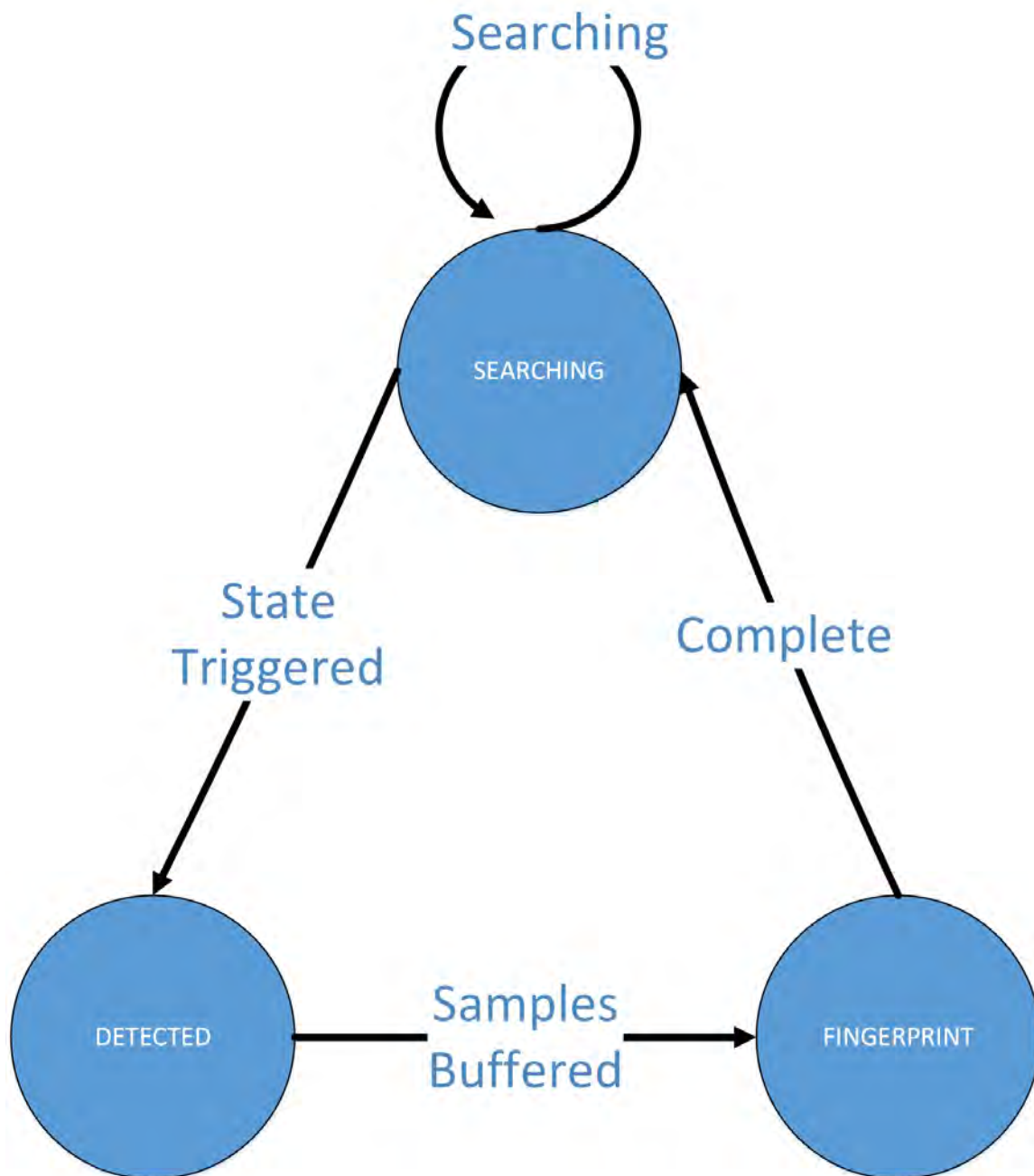


Figure 3.12. *Dna detector ccf* Block State Diagram

Range rx_gain, *QT GUI Time Sink*, and *QT GUI Histogram Sink* blocks as demonstrated on Figure 3.14.

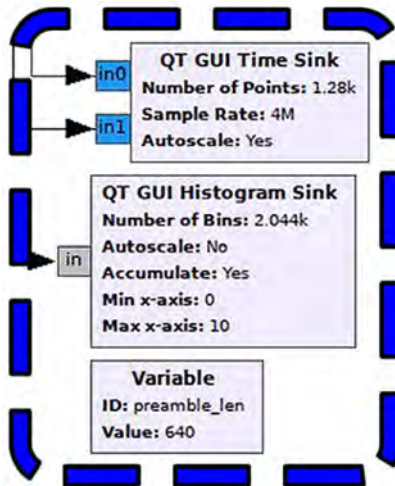


Figure 3.13. Data Visualization Blocks

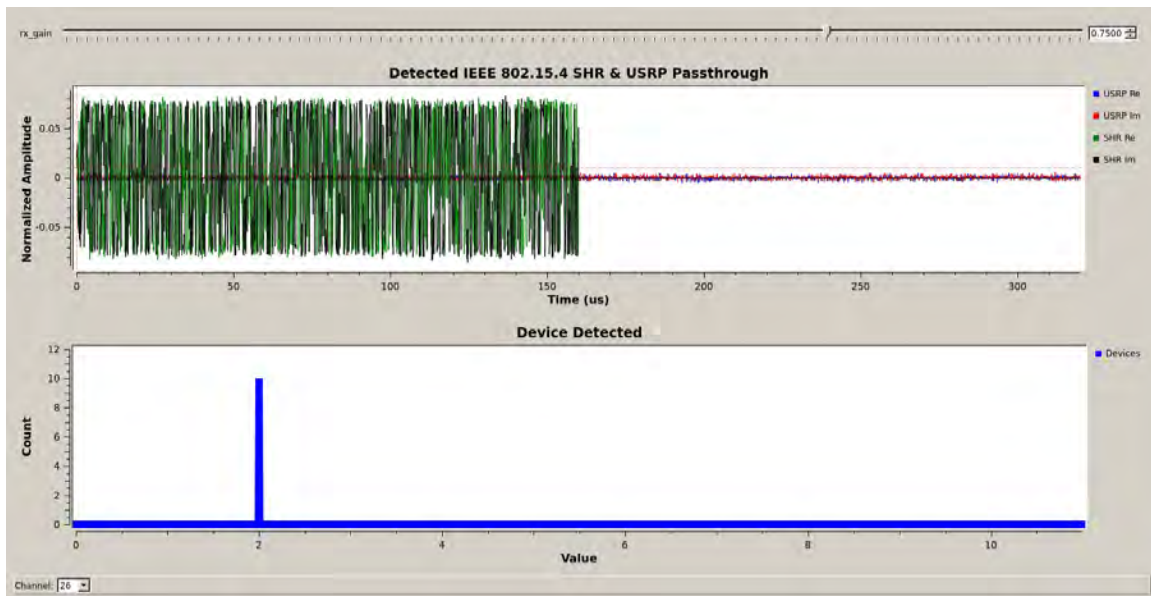


Figure 3.14. Air Monitor Operational GUI

3.1.2.6 Data Storage.

The *File Sink* blocks on Figure 3.15 stored binary complex data received from *preamble_sink* and binary byte data from the *Wireshark Connector* block. All of the collected data was set to store in the root directory. The complex data *File Sink* blocks were only enabled when fingerprints were being stored for off-line analysis. Each filename contained the device, channel number, and source as a complex float binary file as

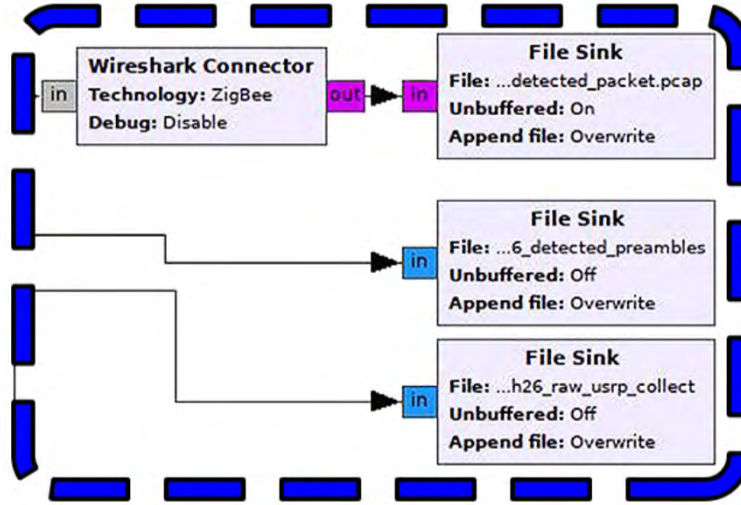


Figure 3.15. Data Storage Blocks

"*Emitter##_Ch26_detected_preambles*" and "*Emitter##_Ch26_raw_usrp_collect*". Similarly, each IEEE 802.15.4 PSDU packet the *Wireshark Connector* block was received from the *preamble_sink* block were converted into a libpcap file. The libpcap files followed the same naming convention where the device and channel collected on were stored as "*Emitter##_Ch26_detected_packet.pcap*".

3.2 Burst Detection

As mentioned in Section 3.1.2.4, each burst is captured and detected within the *preamble_sink* block. Burst detection begins once an IEEE 802.15.4 PSDU packet is fully processed within the the *STATE_SYNC_SEARCH*. The detection process is initiated by acquiring the largest instantaneous amplitude ($\mathbf{a}$) within the temporary buffer $\mathbf{x}[n]$ containing the captured samples from the *STATE_SYNC_SEARCH* state as

$$C_{\text{Max}} = \text{Max}(\mathbf{a}[n]), \quad n = \{0, 1, 2, \dots, N - 1\} \quad (3.5)$$

to create a normalized instantaneous amplitude $\mathbf{a}[n]_{\text{Norm}}$ by

$$\mathbf{a}[n]_{\text{Norm}} = \frac{\mathbf{a}[n]}{C_{\text{Norm}}}, \quad n = \{0, 1, 2, \dots, N-1\}. \quad (3.6)$$

The burst detection threshold $t_D = 0.3$ established by *Burst Trigger Percentage* in the *preamble_sink* block was used detect the index point N_{Index} where

Algorithm 1 Threshold Detection

```

function BURST_TRIGGER( $\mathbf{a}[n]_{\text{Norm}}, t_D, N_{\text{Index}}$ )
     $n = 0$ 
     $N = \text{length}(\mathbf{a}[n]_{\text{Norm}})$ 
    while  $n < N$  do
        if  $\mathbf{a}[n]_{\text{Norm}} \geq t_D$  then
             $N_{\text{Index}} = n$ 
            break
        end if
         $n = n + 1$ 
    end while
    return  $N_{\text{Index}}$ 
end function

```

N_{Index} and the number of samples within an IEEE 802.15.4 SHR N_{SHR} were used to determine if $\mathbf{x}[n]$ contains enough samples to properly create a detected SHR sequence $\mathbf{x}[n]_{\text{Detected}}$ by Algorithm 2.

The temporary buffer $\mathbf{x}[n]$ containing the detected burst is composed of approximately $N \approx 10,220$ samples when the *Buffer Length* is configured to store 4 bursts and 2,044 samples from the most recent burst. At $R_{\text{Samp}} = 4$ MSamp/sec, the burst detector will search for $N_{\text{SHR}} = 640$ samples within $\mathbf{x}[n]$. Figure 3.16 contains a complete normalized IEEE 802.15.4 burst triggered at $t_D = 0.3$ and the detected SHR burst.

The purpose of normalizing the IEEE 802.15.4 SHR burst is to establish a starting point for burst detection. The unnormalized complex IEEE 802.15.4 SHR samples rendered on Figure 3.17 will be fingerprinted in the *Dna detector ccf* block.

Algorithm 2 Burst Detection

```
function BURST DETECTION( $\mathbf{x}[n]$ ,  $N_{\text{Index}}$ ,  $N_{\text{SHR}}$ ,  $\mathbf{x}[n]_{\text{Detected}}$ )  
     $n = 0$   
     $N = \text{length}(\mathbf{x}[n])$   
    if ( $N_{\text{Index}} + N_{\text{SHR}} + 1 > N$ ) then  
        while  $n < N$  do  
             $\mathbf{x}[n] = 0$   
             $n = n + 1$   
        end while  
    else if ( $N_{\text{Index}} + N_{\text{SHR}} + 1 \leq N$ ) then  
         $n = N_{\text{Index}}$   
        while  $n < (N_{\text{Index}} + N_{\text{SHR}} + 1)$  do  
             $\mathbf{x}[n]_{\text{Detected}} = \mathbf{x}[n]$   
             $n = n + 1$   
        end while  
        while  $n < N$  do  
             $\mathbf{x}[n]_{\text{Detected}} = 0$   
             $n = n + 1$   
        end while  
    end if  
    return  $\mathbf{x}[n]_{\text{Detected}}$   
end function
```

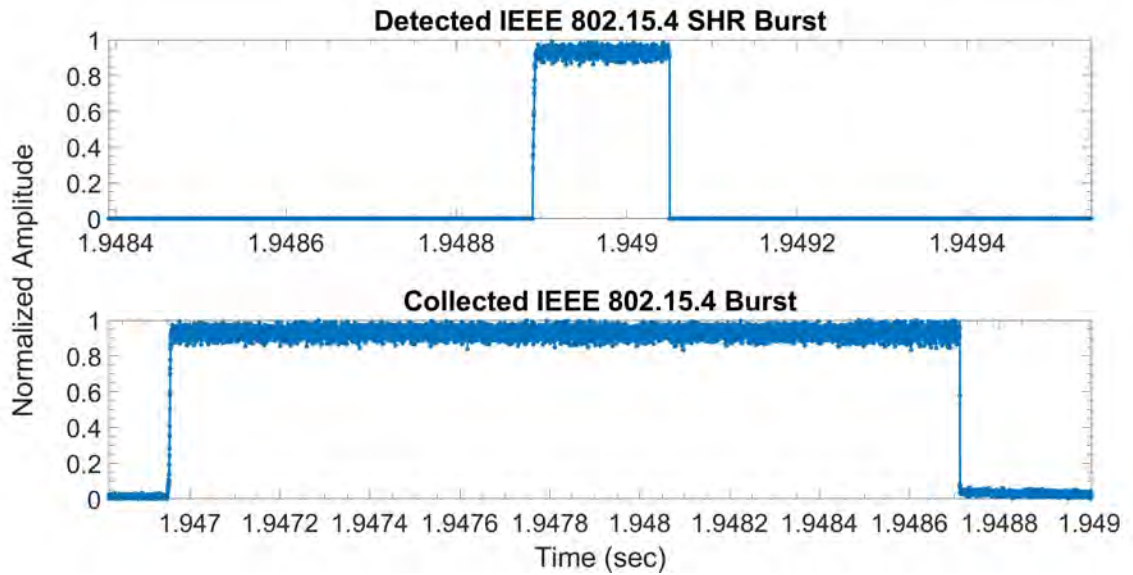


Figure 3.16. Normalized IEEE 802.15.4 Packet and SHR Bursts from Device 10

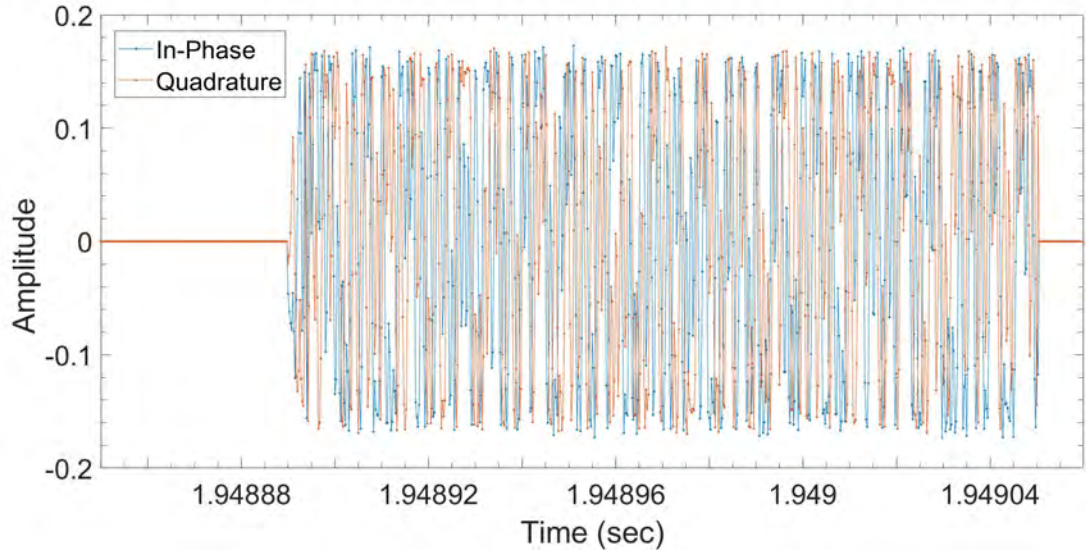


Figure 3.17. Detected IEEE 802.15.4 SHR Burst from Device 10

3.3 Fingerprint Generation

Full-dimensional Time Domain (TD) fingerprints $\mathbf{F}_{\text{TD}}$ generated in this research with MATLAB and the *Dna detector ccf* block were $N_{\text{TD}} = 90$ samples long from each captured IEEE 802.15.4 SHR waveform samples. Each full-dimensional TD fingerprint $\mathbf{F}_{\text{TD}}$ was comprised of $N_R = 10$ Region(s) of Interest (ROI), $N_F = 3$ features per ROI, and $N_M = 3$ statistical metrics per feature as listed in Section 2.4 such that

$$\mathbf{F}_{\text{TD}} = \left[\mathbf{F}_1^a : \mathbf{F}_1^\phi : \mathbf{F}_1^f : \dots : \mathbf{F}_{10}^a : \mathbf{F}_{10}^\phi : \mathbf{F}_{10}^f \right]_{[1 \times 3 \times 3 \times 10] = [1 \times 90]}. \quad (3.7)$$

The MATLAB generated fingerprints were used specifically to create an estimative MDA/ML_{Est} model simulating an environment with a variable Signal-to-Noise Ratio (SNR) to compare against the air monitor's fingerprint trained MDA/ML_{Air} model.

The average signal power S and average noise power N were used to calculate the SNR as

$$\text{SNR} = 10 \log_{10} \left(\frac{S}{N} \right), \quad (3.8)$$

where the average power of a complex signal $\mathbf{x}[n]$ is calculated with it's complex conjugate $\mathbf{x}^*[n]$ is given by

$$X = \frac{1}{N} \sum_{n=0}^{N-1} \mathbf{x}[n] \mathbf{x}^*[n]. \quad (3.9)$$

The captured IEEE 802.15.4 signal $\mathbf{s}[m]_c$ is composed from the collected transmitted signal $\mathbf{s}[m]$ and collected background noise $\mathbf{n}[m]_b$,

$$\mathbf{s}[m]_c = \mathbf{s}[m] + \mathbf{n}[m]_b, \quad m = \{0, 1, 2, \dots, M-1\}, \quad (3.10)$$

such that the captured signal $\mathbf{s}[m]_c$ average power S_c , collected transmitted signal $\mathbf{s}[m]$ average power S , and collected background noise $\mathbf{n}[m]_b$ average power N_b is calculated as

$$\begin{aligned} S_c &= S + N_b, \\ S_c &= \frac{1}{M} \sum_{m=0}^{M-1} \mathbf{s}[m]_c \mathbf{s}^*[m]_c, \\ S &= \frac{1}{M} \sum_{m=0}^{M-1} \mathbf{s}[m] \mathbf{s}^*[m], \\ N_b &= \frac{1}{M} \sum_{m=0}^{M-1} \mathbf{n}[m]_b \mathbf{n}^*[m]_b, \end{aligned} \quad (3.11)$$

where the collected background noise $\mathbf{n}_b[m]$ was measured where there were no signals present. The MATLAB modeled fingerprints added scaled Additive White Gaussian Noise (AWGN) noise $\mathbf{n}_{\text{AWGN}[m]}$ from SNR = 0 to 40 dB in 5dB increments. Sections 3.3.1 and 3.3.2 will explain how each set of fingerprints were created.

3.3.1 MATLAB Fingerprint Generation.

The data originally collected was created from a Pseudo Random Noise Generated (PRNG) sequence and stored in a text file. The text file was converted to a libpcap

file using *zbconvert* so that IEEE 802.15.4 bursts could be generated using *zbreplay*. Each RZUSBstick transmitted $N_{\text{Bursts}} = 12,000$ bursts on channel $f_{Ch} = 26$ ($f_C = 2.48$ GHz) at $T_{\text{Burst}} = 50$ msec intervals. The RZUSBstick transmitted inside a Ramsey test enclosure, and the signal was collected at a sample rate of $R_{\text{Samp}} = 10$ MSamp/sec. The burst detection computed moving average of the energy of the collected signal, and determined the presence of a burst once the moving average exceeded a specified threshold.

The data used to generate the MATLAB fingerprints was collected for each RZUSBstick device before development on the air monitor began. The air monitor captured data at a sample rate of $R_{\text{Samp}} = 4$ MSamp/sec per the design of the original *IEEE 802.15.4 O-QPSK PHY* transceiver block. The sample rate was kept at $R_{\text{Samp}} = 4$ MSamp/sec to ensure that the air monitor's original receive functionality did not deviate from the intended design. The previously collected data had to be resampled from $R_{\text{Samp}} = 10$ MSamp/sec to $R_{\text{Samp}} = 4$ MSamp/sec before the MATLAB fingerprints could be modeled with AWGN.

The first step to properly resample the off-line processed data was to up-sample the collected signal $\mathbf{x}[n]$ by a factor of 2 for a resulting sampling rate of $R_{\text{Samp}} = 20$ MSamp/sec by interpolating zeroes. Figure 3.18 is a normalized burst from the collected signal $\mathbf{x}[n]$ at $R_{\text{Samp}} = 10$ MSamp/sec and Figure 3.19 is the up-sampled signal $\mathbf{x}[n]_{\text{Up}}$ at $R_{\text{Samp}} = 20$ MSamp/sec.

Then the up-sampled signal $\mathbf{x}[n]_{\text{Up}}$ is filtered with a 10-order low pass filter to mitigate the artifacts introduced by the up-sampling process. The upsampled-filtered signal $\mathbf{x}[n]_{\text{Filt}}$ represented on Figure 3.20 is finally down-sampled by a factor of 5 for a resulting sample rate of $R_{\text{Samp}} = 4$ MSamp/sec as shown on Figure 3.21

After the collected signal is re-sampled to $R_{\text{Samp}} = 4$ MSamp/sec, independent AWGN zero-mean, like-filtered noise samples $\mathbf{n}[m]_{\text{AWGN}}$ was added to the resampled

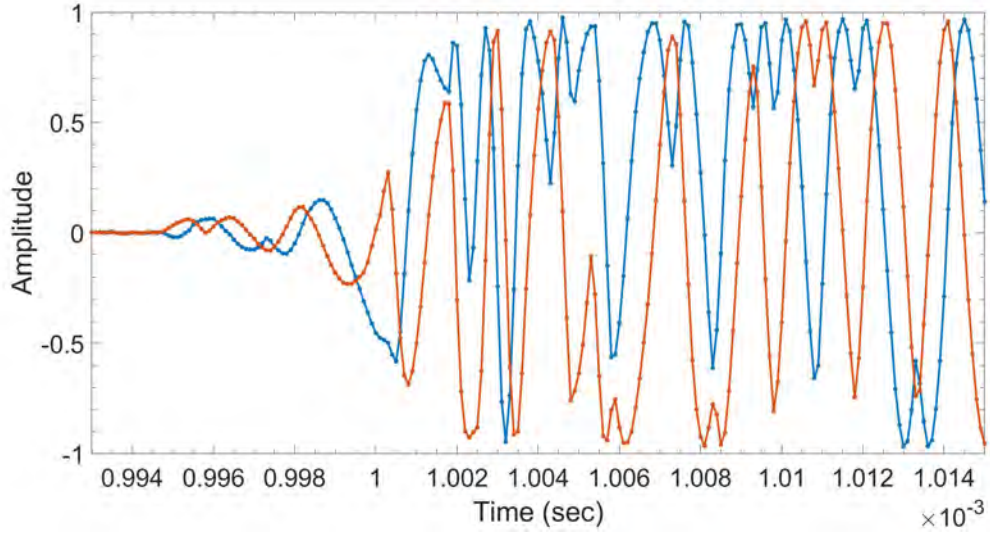


Figure 3.18. Normalized Burst at $R_{\text{Samp}} = 10$ MSamp/sec from Device 10

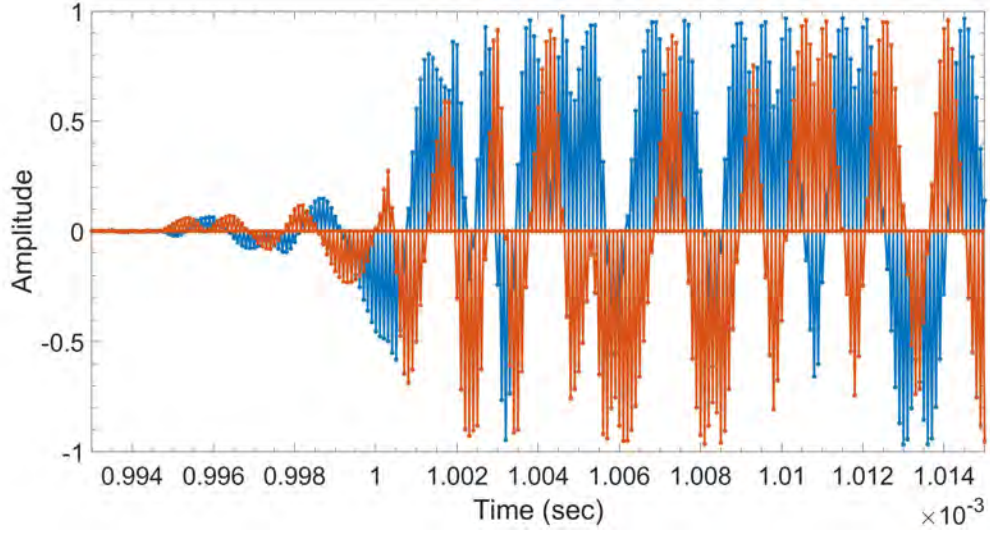


Figure 3.19. Normalized Burst at $R_{\text{Samp}} = 20$ MSamp/sec from Device 10

data $\mathbf{x}[m]_{\text{Resamp}}$ as

$$\mathbf{x}[m]_{\text{Model}} = \mathbf{x}[m]_{\text{Resamp}} + \mathbf{n}[m]_b + \mathbf{n}[m]_{\text{AWGN}}, \quad m = \{0, 1, 2, \dots, M-1\}, \quad (3.12)$$

where the average power of $\mathbf{n}[m]_{\text{AWGN}}$ is scaled to achieve an SNR = 0 to 40 dB [9].

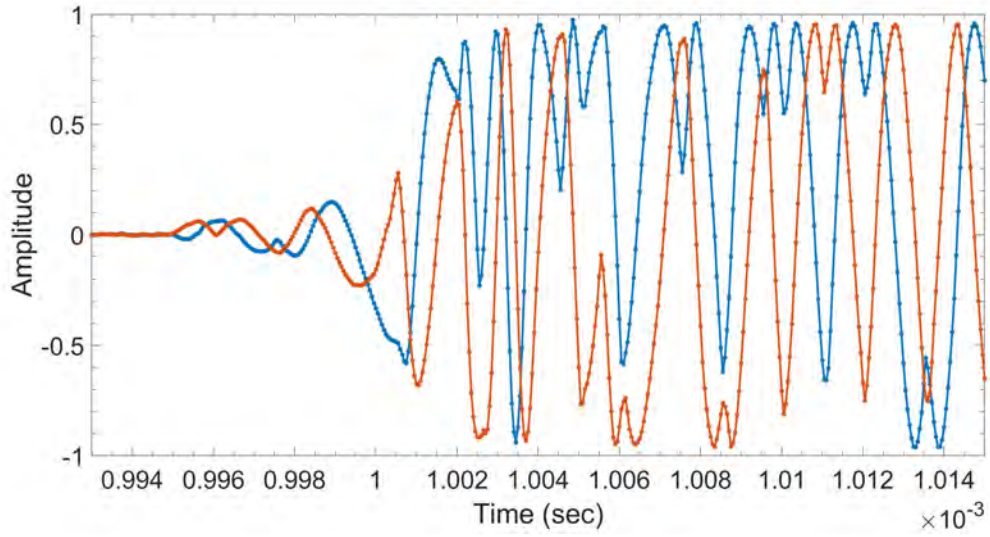


Figure 3.20. Filtered Normalized Burst at $R_{\text{Samp}} = 20 \text{ MSamp/sec}$ from Device 10

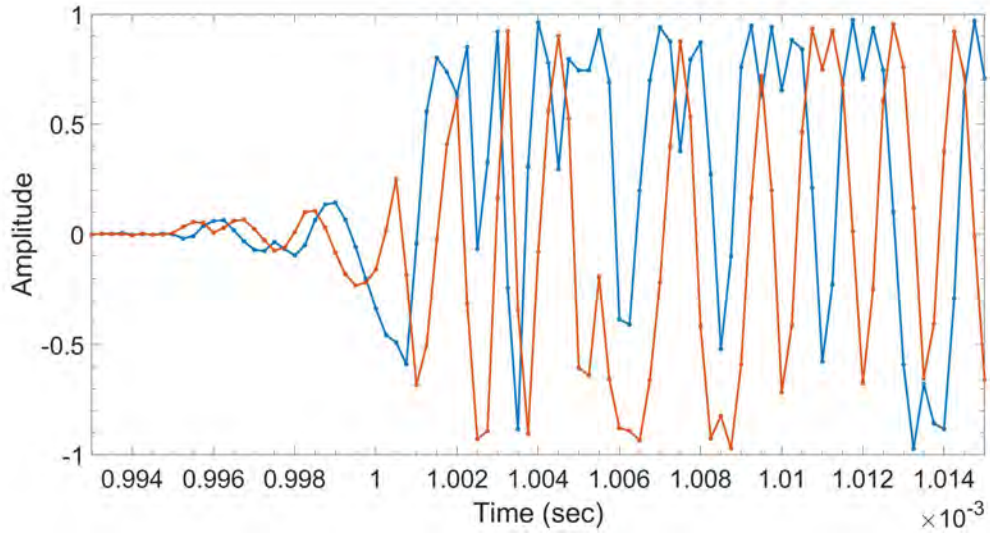


Figure 3.21. Normalized Burst at $R_{\text{Samp}} = 4 \text{ MSamp/sec}$ from Device 10

Each noise sample sequence $\mathbf{n}[m]_{\text{AWGN}}$ has an estimated average power of $N = 1 \text{ dB}$ that is power scaled by the downsampled signals average power X_{Resamp} and a desired SNR such that

$$A_{\text{AWGN}} = \sqrt{10^{\frac{-\text{SNR}_{\text{Desired}}}{10}} \times X_{\text{Resamp}}}, \quad (3.13)$$

where A_{AWGN} is a noise scaling multiplier used to calculate the total average desired noise power N_{AWGN} as

$$N_{\text{AWGN}} = \frac{1}{M} \sum_{m=0}^{M-1} A_{\text{AWGN}} \mathbf{n}[m]_{\text{AWGN}} A_{\text{AWGN}} \mathbf{n}^*[m]_{\text{AWGN}}. \quad (3.14)$$

The resampled collection $\mathbf{x}[n]_{\text{Resamp}}$ will add the total average desired noise power N_{AWGN} from an SNR = 0 to 40 dB in 5 dB increments into the SNR modeled samples $\mathbf{x}[n]_{\text{Model}}$ as

$$X_{\text{Model}} = X_{\text{Resamp}} + N_b + N_{\text{AWGN}}, \quad (3.15)$$

before each burst is fingerprinted. The SNR for each modeled sample collection $\mathbf{x}[n]_{\text{Model}}$ can be calculated by extending (3.8) using

$$\text{SNR} = 10 \log_{10} \left(\frac{X_{\text{Model}}}{N_b + N_{\text{AWGN}}} \right). \quad (3.16)$$

3.3.2 GNU Radio Fingerprint Generation.

Each RZUSBstick was programmed to replay *control4-sample.pcap*, a libpcap containing ZigBee data from a Control4[®] appliance. The libpcap file transmitted $N_{\text{Bursts}} = 239$ bursts on channel $f_{Ch} = 26$ ($f_C = 2.48$ GHz) at $T_{\text{Burst}} = 10$ msec intervals between bursts. A bash script was created to consecutively replay the file 8-times with *zbreplay* to generate $N_{\text{Bursts}} = 1,912$ bursts on each device. More information will be provided on about the transmitted IEEE 802.15.4 data packets in Chapter IV. Figure 3.22 is one iteration of *control4-sample.pcap* on device 10.

The captured signals $\mathbf{x}[n]_c$ SNR was calculated using

$$\text{SNR} = 10 \log_{10} \left(\frac{X_c - N_b}{N_b} \right) = 10 \log_{10} \left(\frac{(X + N_b) - N_b}{N_b} \right) \approx 10 \log_{10} \left(\frac{X}{N_b} \right) \quad (3.17)$$

where the average background noise power N_b is assumed to be captured within the

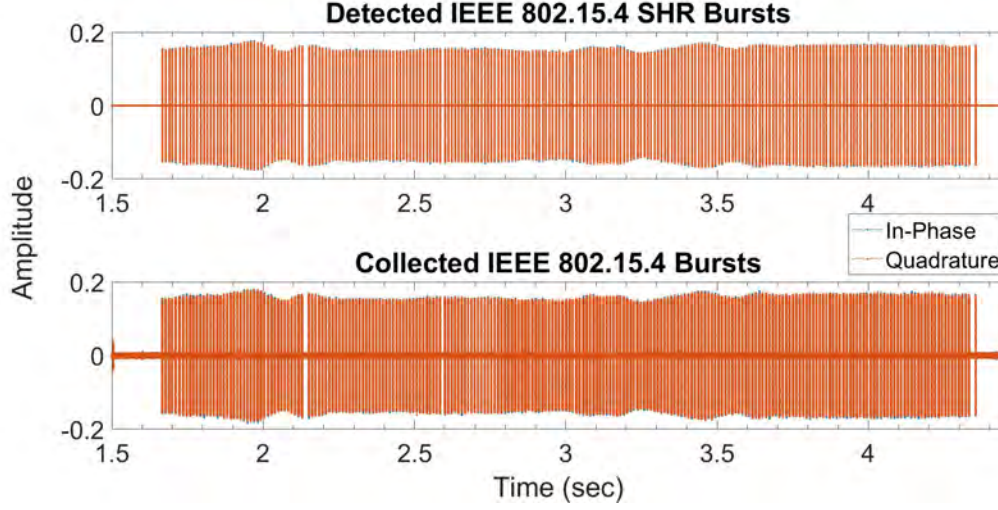


Figure 3.22. $N_{\text{Bursts}} = 239$ Bursts from *control4-sample.pcap* on Device 10

average captured signal's power X_c , so the measured background noise power N_b was removed from captured signals power X_c to get an approximate value for the average transmitted signals average power X . The captured signals $\mathbf{x}[n]_c$ SNR and fingerprint results will be covered in Chapter IV.

3.4 MDA/ML

The MATLAB and air monitor generated fingerprints for the $N_D = 10$ RZUSBsticks were used for MDA model development and Maximum Likelihood (ML) device classification. $N_F = 1,000$ fingerprints were taken for each set of SNR augmented bursts within MATLAB to generate an MDA/ML_{Est} model to formulate a comparative baseline for the air monitor at various SNR's. $N_F = 1,000$ fingerprints that were created by the air monitor were loaded into MATLAB to create an MDA/ML_{Air} model at the measured SNR. That air monitor's MDA_{Air} model was loaded into the *DNA detector ccf* block afterwards to discriminate devices in NRT.

MDA/ML processing was performed as described in Section 2.5. The process begins with each set of $N_F = 1,000$ fingerprints being separated into $N_{\text{Train}} = 500$ training

and $N_{\text{Test}} = 500$ testing fingerprint sets. Both N_{Train} and N_{Test} fingerprint sets were taken as interleaved subsets of the fingerprint set. The N_{Train} fingerprints were then fed into the MDA/ML classifiers where the model were created. A K-fold cross-validation processed was to used determine an ideal model for the given set with $K = 5$ -folds. The developed models performance was measured by performing a “looks most like” assessment on each N_{Test} fingerprint set and assigning each set to the postulated device [9].

The air monitors MDA_{Air} model data consisted of a

- $\mathbf{W}_{[90 \times 9]}$ - MDA projection matrix
- $\mathbf{W}_{\mu[10 \times 9]}$ - Trained MDA class mean device projection matrix
- $\mathbf{F}_{\text{TD Norm}[1 \times 90]}$ - Fingerprint normalization gain vector
- $\mathbf{F}_{\text{TD XOffset}[1 \times 90]}$ -Fingerprint offset vector

where the $\mathbf{F}_{\text{TD Norm}}$ and $\mathbf{F}_{\text{TD XOffset}}$ must be applied to the *Dna detector ccf* block's $\mathbf{F}_{\text{NRT}}$ fingerprint because the MDA_{Air} model's values were normalized and offset in MATLAB to maintain a maximum dynamic range for the floating point representations. The offset and gain normalized fingerprint $\mathbf{F}_{\text{TD}}$ is represented by

$$\mathbf{F}_{\text{TD}} = (\mathbf{F}_{\text{NRT}} - \mathbf{F}_{\text{TD XOffset}}) \circ \mathbf{F}_{\text{TD Norm}}. \quad (3.18)$$

$\mathbf{F}_{\text{TD}}$ is then applied to a modified version of (2.23) so that

$$\hat{\mathbf{f}} = \mathbf{F}_{\text{TD}} \mathbf{W} \quad (3.19)$$

where the resultant projection vector $\hat{\mathbf{f}}$ computed the same results as (2.23), but the output dimensions were 1×9 instead of 9×1 .

Finally, the resultant projection vector $\hat{\mathbf{f}}$ is used to estimate the "most likely" device by calculating the minimum for Euclidean distance $\mathbf{d}_{\text{Min}}$ between the projection vector $\hat{\mathbf{f}}$ and each vector row within $\mathbf{W}_\mu$ as

$$\begin{pmatrix} \left[\sqrt{(\mathbf{w}_{\mu_{1,1}} - \hat{\mathbf{f}}_{1,1})^2 + \dots + (\mathbf{w}_{\mu_{1,9}} - \hat{\mathbf{f}}_{1,9})^2} \right] \\ \vdots \\ \left[\sqrt{(\mathbf{w}_{\mu_{10,1}} - \hat{\mathbf{f}}_{10,1})^2 + \dots + (\mathbf{w}_{\mu_{10,9}} - \hat{\mathbf{f}}_{10,9})^2} \right] \end{pmatrix} = \begin{pmatrix} d_{1,1} \\ \vdots \\ d_{1,10} \end{pmatrix}, \quad (3.20)$$

$$\mathbf{d}_{\text{Min}} = \min(d_{1,m}) \quad \text{for } m = \{1, \dots, 10\},$$

where the row number of the Euclidean distance vector $\mathbf{d}$ with the minimum Euclidean distance $\mathbf{d}_{\text{Min}}$ corresponds to the "most likely" device. The results of discriminating between devices in NRT will be discussed in Chapter IV.

IV. Analysis of Results

This chapter provides the results for ZigBee device discrimination in Near Real-Time (NRT) with the GNU Radio air monitor, to include comparing predictive performance *Device Classification* estimations against live results using Radio Frequency-Distinct Native Attribute (RF-DNA) Fingerprints generated and collected in NRT. This chapter is organized as follows: Section 4.1 provides details on the GNU Radio air monitor NRT fingerprinting performance while simultaneously storing collection data. Section 4.2 provides details on the GNU Radio air monitor operational performance after Multiple Discriminant Analysis/Maximum Likelihood (MDA/ML) trained data has been loaded and data storage was disabled. This is followed by Section 4.3 where an estimative Additive White Gaussian Noise (AWGN) MATLAB and air monitor trained MDA/ML models will be created. Section 4.4 concludes Chapter IV by providing the device discrimination results of the GNU Radio air monitor.

4.1 GNU Radio Air Monitor Collection Performance Analysis

The air monitor's collection performance was assessed based on how well the Software-Defined Radio(s) (SDR) detected Institute of Electrical and Electronics Engineers (IEEE) 802.15.4 bursts from each device; how well the *preamble_sink* block extracted IEEE 802.15.4 Synchronization Header (SHR) waveform samples from the detected bursts; and how well the *Dna_detector_ccf* block fingerprinted and discriminated devices from the extracted SHR samples from the *preamble_sink* block while simultaneously storing data. The air monitor had each complex file sink storing complex data

at

$$\begin{aligned}
R_{\text{Complex}} &= R_{\text{Samp}} \times N_{\text{Complex Size}} \times N_{\text{Byte Size}}, \\
R_{\text{Complex}} &= (4 \text{ MSamp/sec})(8 \text{ B/Samp})(8 \text{ Bits/B}), \\
R_{\text{Complex}} &= (32 \text{ MB/sec})(8 \text{ Bits/B}) = 256 \text{ MBits/sec},
\end{aligned} \tag{4.1}$$

and the byte file sink storing libpcap data at

$$\begin{aligned}
R_{\text{Libpcap}} &= R_{\text{Samp}} \times N_{\text{Libpcap Size}} \times N_{\text{Byte Size}}, \\
R_{\text{Libpcap}} &= (4 \text{ MSamp/sec})(1 \text{ B/Samp})(8 \text{ Bits/B}), \\
R_{\text{Libpcap}} &= (4 \text{ MB/sec})(8 \text{ Bits/B}) = 32 \text{ MBits/sec}.
\end{aligned} \tag{4.2}$$

This means the air monitor was storing data at $R_{\text{Store}} = 68 \text{ MB/sec}$ ($2 \times R_{\text{Complex}} + R_{\text{Libpcap}}$), not including the 3 statistical files, the binary fingerprint file being written, the RZUS-Bsticks transmitting bursts, and background computer processes.

The original libpcap file *control4-sample.pcap* contained $N_{\text{Packets}} = 407$ packets, but only $N_{\text{Packets}} = 239$ packets were replayed at $T_{\text{Burst}} = 10 \text{ msec}$ intervals by *zbreplay* without the acknowledgement (ACK) packets. Table 4.1 is an illustration displaying the minimum, maximum, and average byte length of each IEEE 802.15.4 data packet within *control4-sample.pcap*. The libpcap table consists of:

- - *Packets* - The overall packet count and statistical data for $N_{\text{Total Packets}}$ packets regardless of byte length in the libpcap file
- - *0-19* - The packet count and statistical data for $N_{0-19 \text{ Packets}}$ packets that are between 0 to 19 bytes long. The overall percentage of packets between 0 to 19 bytes long is $\frac{N_{0-19 \text{ Packets}}}{N_{\text{Total Packets}}}$
- - *20-39* - The packet count and statistical data for $N_{20-39 \text{ Packets}}$ packets that are between 20 to 39 bytes long. The overall percentage of packets between 20 to 39

bytes long is $\frac{N_{20-39 \text{ Packets}}}{N_{\text{Total Packets}}}$

- - 40-79 - The packet count and statistical data for $N_{40-79 \text{ Packets}}$ packets that are between 40 to 70 bytes long. The overall percentage of packets between 40 to 79 bytes long is $\frac{N_{40-79 \text{ Packets}}}{N_{\text{Total Packets}}}$
- - 80-159 - The packet count and statistical data for $N_{80-159 \text{ Packets}}$ packets that are between 80 to 159 bytes long. The overall percentage of packets between 80 to 159 bytes long is $\frac{N_{80-159 \text{ Packets}}}{N_{\text{Total Packets}}}$

Table 4.1. Composition of *control4-sample.pcap* With and Without ACK Packets

Original IEEE 802.15.4 Libpcap Data With ACK Packets						
Packet Lengths			Average Val (B)	Min Val(B)	Max Val (B)	Packet Count
	Total Packets	100.00%	36.44	5	90	407
	0-19	43.24%	5.33	5	18	176
	20-39	1.47%	26.67	21	28	6
	40-79	40.54%	51.87	45	73	165
	80-159	14.74%	86.27	80	90	60
Original IEEE 802.15.4 Libpcap Data Without ACK Packets						
Packet Lengths			Average Val (B)	Min Val(B)	Max Val (B)	Packet Count
	Total Packets	100.00%	58.55	10	90	239
	0-19	3.35%	12.25	10	18	8
	20-39	2.51%	26.67	21	28	6
	40-79	69.04%	51.87	45	73	165
	80-159	25.10%	86.27	80	90	60

Each packet's physical signal burst period T_{Packet} can be approximately calculated using the packet byte length N_{Packet} , the SHR byte length N_{SHR} , the PHY Header (PHR) byte length N_{PHR} , and IEEE 802.15.4 Offset-Quadrature Phase Shift Keying (O-QPSK) bit rate

R_b as

$$\begin{aligned}
T_{\text{Packet}} &\approx \frac{(N_{\text{Packet}} + N_{\text{SHR}} + N_{\text{PHR}})(N_{\text{Byte Size}})}{R_b}, \\
T_{\text{Packet}} &\approx \frac{(N_{\text{Packet}} + 5 \text{ B} + 1 \text{ B})(8 \text{ Bits/B})}{250 \text{ kBits/sec}}, \\
T_{\text{Packet}} &\approx \frac{(N_{\text{Packet}} + 6 \text{ B}) \text{ sec}}{31.25 \text{ kB}},
\end{aligned} \tag{4.3}$$

where the longest single burst period for a ZigBee device with a maximum packet length of $N_{\text{Packet}} = 127$ bytes would be $T_{\text{Packet}} \approx 4.26$ msec per the IEEE 802.15.4 standard. According to Table 4.1, the packets transmitted from the libpcap file varied in length from $N_{\text{Packet}} = 10$ to 90 bytes. This caused the transmitted burst-to-burst replay period to shift between $10.51 \text{ msec} \leq T_{\text{TX Burst}} \leq 13.07 \text{ msec}$ due to the transmitted packets physical signal burst period varying anywhere between $0.51 \text{ msec} \leq T_{\text{Packet}} \leq 3.07 \text{ msec}$.

A bash script was created to consecutively replay the libpcap file 8-times to generate $N_{\text{Bursts}} = 1,912$ bursts on each device. The file replay interval T_{Replay} was dependent the operating system's current workload. The period varied anywhere between $0.179 \text{ sec} \leq T_{\text{Replay}} \leq 0.61 \text{ sec}$, but generally kept around $T_{\text{Replay}} \approx 0.58 \text{ sec}$ between each successive replay. Refer to Appendix A for In-Phase and Quadrature-Phase (I/Q) and power plot collection figures for each device. The ideal reconstructed packet composition for $N_{\text{Total Packets}} = 19,120$ packets on Table 4.2 is a baseline metric used to compare the detected data's segregated packet length segments cumulative ratios against. The Absolute Reconstructed Packet Ratio Deviation ($\%A_{\Delta}$) between each detected packet lengths cumulative ratio and the ideal packet lengths cumulative ratio listed on Table 4.2 as

$$\%A_{\Delta} = |\%_{\text{Detected Packet Ratio}} - \%_{\text{Ideal Detected Packet Ratio}}| \tag{4.4}$$

to see how much each packet length segment deviated from the ideal composition.

Table 4.2. Ideal Composition of $N_{\text{Total Packets}} = 19,120$ Detected Packets

Ideal Composition of Reconstructed IEEE 802.15.4 Libpcap Data Without ACK Packets						
Packet Lengths			Average Val (B)	Min Val(B)	Max Val (B)	Packet Count
	Total Packets	100.00%	58.55	10	90	19120
	0-19	3.35%	12.25	10	18	640
	20-39	2.51%	26.67	21	28	480
	40-79	69.04%	51.87	45	73	13200
	80-159	25.10%	86.27	80	90	4800

The overall statistical data collected from all the RZUSBstick's transmitting $N_{\text{Overall}} = 19,120$ overall bursts is below on Table 4.3. Table 4.3 contains the minimum, maximum, and average of the overall collections results listed below as::

- *Time (sec)* - Timestamp when the first and last recovered packet were captured in the libpcap file.
- *Packets Detected* - The packet count and statistical data for N_{Detected} packets for all of the detected packets within the *preamble_sink* block. The Overall Packet Detection Percent (%D) is $\frac{N_{\text{Detected}}}{N_{\text{Overall}}}$
- *Short Packets* - The packet count and statistical data for N_{Short} packets for detected packets with frame lengths shorter than minimum frame length of 9 bytes within the *preamble_sink* block (See Figure 2.3). The overall percentage of short packets is $\frac{N_{\text{Short}}}{N_{\text{Overall}}}$
- *Bad Cyclic Redundancy Check (CRC) Packets* - The packet count and statistical data for $N_{\text{CRC Bad}}$ packets for detected packets that contain data errors detected during a CRC-16 check within the *preamble_sink* block. The overall percentage of packets that failed the CRC-16 data check is $\frac{N_{\text{CRC Bad}}}{N_{\text{Overall}}}$
- *Good CRC Packets* - The packet count and statistical data for $N_{\text{CRC Good}}$ packets for detected packets that passed the CRC-16 check within the *preamble_sink* block.

The Overall Accurate Packet Reconstruction Percent ($\%R$) that passed the CRC-16 data check is $\frac{N_{\text{CRC Good}}}{N_{\text{Overall}}}$

- *Preambles Accepted* - The burst count and statistical data for N_{Accepted} bursts for detected IEEE 802.15.4 SHR bursts within the *preamble_sink* block. The overall percentage of detected IEEE 802.15.4 SHR bursts is $\frac{N_{\text{Accepted}}}{N_{\text{Overall}}}$
- *Preambles Rejected* - The burst count and statistical data for N_{Rejected} bursts for rejected IEEE 802.15.4 SHR bursts within the *preamble_sink* block. The overall percentage of rejected IEEE 802.15.4 SHR bursts is $\frac{N_{\text{Rejected}}}{N_{\text{Overall}}}$
- *Total Processed Fingerprints* - The fingerprint count and statistical data for $N_{\text{Fingerprint}}$ fingerprints generated in the *Dna detector ccf* block from detected IEEE 802.15.4 SHR bursts within the *preamble_sink* block. The Overall Fingerprinted Percent ($\%F$) of IEEE 802.15.4 SHR bursts is $\frac{N_{\text{Fingerprint}}}{N_{\text{Overall}}}$
- *% Fingerprints Processed* - The ratio between the $N_{\text{Fingerprint}}$ generated fingerprints within the *Dna detector ccf* block and the accepted IEEE 802.15.4 SHR bursts within the *preamble_sink* block is computed as $\frac{N_{\text{Fingerprint}}}{N_{\text{Accepted}}}$

The collection results of the $N_d = 10$ RZUSBsticks displayed on Table 4.3 shows the packet detection rate $\%D = 97.93\%$ from the $N_{\text{Overall}} = 19,120$ transmitted bursts within an overall average $T_{\text{Collect}} = 25.691$ sec transmission period. $\%R = 97.92\%$ of the IEEE 802.15.4 data packets passed CRC-16 verification and 97.85% IEEE 802.15.4 SHR waveform bursts were detected from the overall transmitted bursts. $\%F = 97.48\%$ of the transmitted bursts were fingerprinted within the *Dna detector ccf* block. 99.63% of the $N_{\text{Accepted}} = 18,708$ accepted IEEE 802.15.4 SHR bursts were fingerprinted and used for device discrimination.

The *Dna detector ccf* block performs device discrimination using invalid data (i.e. identity matrices and vectors of one's) whenever trained data is not uploaded into the

Table 4.3. Overall Collection Performance for $N_d = 10$ Devices

				Avg	Min	Max	Overall Numbers
Time (sec)				25.691	25.534	25.89	
Packets Transmitted			100.00%	1912	1912	1912	19120
Preamble Sink Detected Bursts	Packet Data	Total Packets Detected(D%)	97.93%	1872.5	1820	1910	18725
		Short Packets	0.01%	0.1	0	1	1
		Bad CRC Packets	0.01%	0.1	0	1	1
		Good CRC Packets (R%)	97.92%	1872.3	1820	1910	18723
	SHR	Preambles Accepted	97.85%	1870.8	1820	1909	18708
		Preambles Rejected	0.09%	1.7	0	4	17
Dna detector Fingerprinted SHR	Total Processed Fingerprints(F%)		97.48%	1863.9	1801	1909	18639
	% Preambles Fingerprinted		99.63%	99.63%	98.96%	100.00%	
Reconstructed IEEE 802.15.4 Libpcap Packet Data							
				Average Val (B)	Min Val(B)	Max Val (B)	Overall Packet Count
Packet Lengths		Total Packets	100.00%	58.572	1	90	18725
		0-19	3.33%	12.219	1	18	624
		20-39	2.50%	26.687	21	28	469
		40-79	69.03%	51.877	45	73	12926
		80-159	25.13%	86.269	80	90	4706

block. The air monitor processed a IEEE 802.15.4 SHR waveform once a complete data packet was processed. It generally took the air monitor between $0.46 \text{ msec} \leq T_{\text{MDA}} \leq 1.5 \text{ msec}$ from the time a burst was detected to the time a device was classified.

The detected overall packet length cumulative ratio on Table 4.3 deviated from the ideal values on Table 4.2 on average by $\%A_{\Delta} = 0.0425\%$ for all of the packet lengths. 97.50% ($N_{0-19} = 624$) of the 0 to 19 byte length packets were recovered with an $\%A_{\Delta \text{Diff } 0-19} = 0.02\%$ packet ratio deviation; 97.71% ($N_{20-39} = 469$) of the 20 to 39 byte length packets were recovered with an $\%A_{\Delta \text{Diff } 20-39} = 0.01\%$ packet ratio deviation; 97.92% ($N_{40-79} = 12,926$) of the 40 to 79 byte length packets were recovered with an $\%A_{\Delta \text{Diff } 40-79} = 0.01\%$ packet ratio deviation; and 98.04% ($N_{80-159} = 4,706$) of the 80 to 159 byte length packets were recovered with an $\%A_{\Delta \text{Diff } 80-159} = 0.13\%$ packet ratio deviation. Overall, 99.99% ($\frac{N_{\text{CRC-16}}}{N_{\text{Detected}}}$) of the $N_{\text{Detected}} = 18,725$ detected packets were CRC-16 verified and contained appropriate frame lengths.

4.2 GNU Radio Air Monitor Operational Performance Analysis

The same metrics used in Section 4.1 were utilized to assess the NRT operational performance of the air monitor. The difference in this test is that the air monitor did not store any collected fingerprints as binary data, did not store any complex binary data, and loaded the air monitors fingerprint trained MDA_{Air} into the *Dna detector ccf* block.

The operational results of the $N_d = 10$ RZUSBsticks displayed on Table 4.4 shows the packet detection rate increased to $\%D = 99.99\%$ from the $N_{\text{Overall}} = 19,120$ transmitted bursts within an overall average $T_{\text{Collect}} = 25.1958 \text{ sec}$ transmission period. The overall IEEE 802.15.4 data packets that passed CRC-16 verification displayed an improvement to $\%R = 99.96\%$ and 99.91% of the IEEE 802.15.4 SHR waveform bursts were detected from the overall transmitted bursts. The overall fingerprinted bursts within the *Dna*

Table 4.4. Overall Operational Performance for $N_d = 10$ Devices

Time (sec)				25.1958	25.108	25.309	
Packets Transmitted			100.00%	1912	1912	1912	19120
Preamble Sink Detected Bursts	Packet Data	Total Packets					
		Detected(D%)	99.99%	1911.8	1911	1913	19118
		Short Packets	0.02%	0.4	0	1	4
		Bad CRC Packets	0.01%	0.2	0	1	2
		Good CRC Packets(R%)	99.96%	1911.2	1910	1912	19112
	SHR	Preambles Accepted	99.91%	1910.3	1908	1912	19103
		Preambles Rejected	0.08%	1.5	0	3	15
Dna detector Fingerprinted SHR	Total Processed Fingerprints(F%)		99.88%	1909.7	1907	1912	19097
	% Preambles Fingerprinted		99.97%	99.97%	99.84%	100.00%	
Reconstructed IEEE 802.15.4 Libpcap Packet Data							
				Average Val (B)	Min Val(B)	Max Val (B)	Overall Packet Count
Packet Lengths		Total Packets	100.00%	58.538	1	90	19118
		0-19	3.37%	12.196	1	18	644
		20-39	2.51%	26.67	21	28	480
		40-79	69.02%	51.872	45	73	13195
		80-159	25.10%	86.27	80	90	4799

detector ccf block increased to $\%F = 99.88\%$ during the operational trial. 99.97% of the $N_{Accepted} = 19,103$ accepted IEEE 802.15.4 SHR bursts were fingerprinted and used for device discrimination. The time a burst was detected to the time a device was discriminated was not measured for this test as it was assumed to be $0.45 \text{ msec} \leq T_{MDA} \leq 1.5 \text{ msec}$ or better.

The detected overall packet length cumulative ratio on Table 4.3 compared against the ideal values on Table 4.2 deviated on average by $\%A_{\Delta} = 0.035\%$ for all of the packet lengths during the operational test. 100.63% ($N_{0-19} = 644$) of the 0 to 19 byte length packets were recovered with an $\%A_{\Delta \text{Diff } 0-19} = 0.02\%$ packet ratio deviation; 100.0% ($N_{20-39} = 480$) of the 20 to 39 byte length packets were recovered with an $\%A_{\Delta \text{Diff } 20-39} = 0.02\%$ packet ratio deviation; 99.96% ($N_{40-79} = 13,195$) of the 40 to 79 byte length packets were recovered with an $\%A_{\Delta \text{Diff } 40-79} = 0.01\%$ packet ratio deviation;

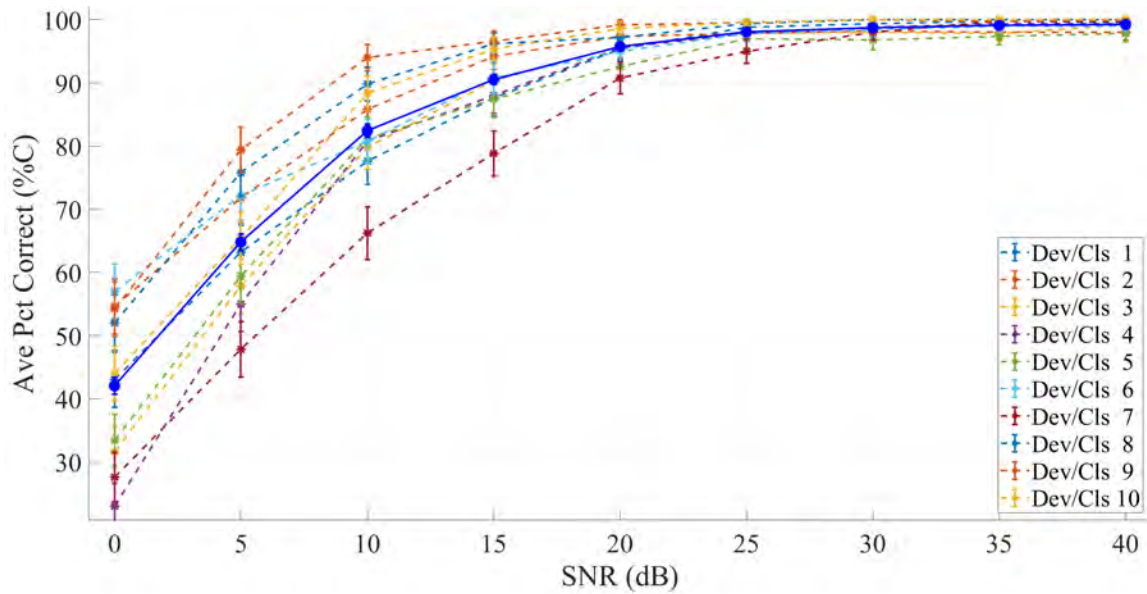
and 99.98% ($N_{80-159} = 4,799$) of the 80 to 159 byte length packets were recovered with an $\%A_{\Delta\text{Diff } 80-159} = 0.1\%$ packet ratio deviation. This experiment resulted in 99.97% ($\frac{N_{\text{CRC-16}}}{N_{\text{Detected}}}$) of the $N_{\text{Detected}} = 19,118$ detected packets were CRC-16 verified with appropriate frame lengths.

4.3 Air Monitor MDA/ML Classification Model Development

An estimative MDA/ML_{Est} performance model between $0 \text{ dB} \leq \text{Signal-to-Noise Ratio (SNR)} \leq 40 \text{ dB}$ for the $N_d = 10$ RZUSBsticks was developed using the methods previously listed in Sections 3.4 and 3.3.1. The Average Percent Correct Classification ($\%C$) is the percent of times that the classifier correctly classified a device. Table 4.5 contains the computed $\%C$ for each set of testing fingerprints within the SNR model. The range was chosen because it was assumed that the air monitor would operate somewhere within the modeled SNR range. The MDA/ML_{Est} model for the $N_d = 10$ devices shows that the $\%C \geq 90.00\%$ at an $\text{SNR} = 15 \text{ dB}$.

After the MATLAB MDA/ML_{Est} model was built, the SNR of the $N_d = 10$ RZUSBsticks was measured before any of the Radio Frequency (RF) air monitor's fingerprints were used for MDA/ML_{Air} processing. The SNR varied between $31 \text{ dB} \leq \text{SNR} \leq 35 \text{ dB}$ with an overall collection mean $\text{SNR} \approx 33.571 \text{ dB}$. Comparing the collected devices SNR to Table 4.5, the air monitors $\%C$ was estimated to be between $98.76\% \leq \%C \leq 99.10\%$. The air monitor fingerprint trained MDA/ML_{Air} model revealed that the $\%C$ varied between $98.20\% \leq \%C \leq 100.00\%$ with an overall $\%C = 99.64$ as shown on the Confusion Matrix (CM) on Table 4.6. The CM contains the $\%C$ of each RZUSBstick on the highlighted diagonal headings while the non-diagonal values contains the Individual % Incorrect Classification ($\%I$) of each misclassified device (i.e. device 1 classified as device 2,3,etc).

Table 4.5. Estimative RF-DNA MDA/ML Cross Classification Performance Model from $0 \text{ dB} \leq \text{SNR} = 40 \text{ dB}$



SNR (dB)	0	5	10	15	20	25	30	35	40
Testing % Correct	42.04%	64.78%	82.44%	90.54%	95.76%	98.06%	98.76%	99.10%	99.28%

Table 4.6. Confusion Matrix $N_d = 10$ Class Problem at $\text{SNR} \approx 33.571 \text{ dB}$

Collected Model	Classified Device (%)										
	Device	1	2	3	4	5	6	7	8	9	10
Input Device (%)	1	99.40%	0.00%	0.00%	0.60%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
	2	0.00%	98.20%	0.00%	0.00%	1.80%	0.00%	0.00%	0.00%	0.00%	0.00%
	3	0.00%	0.00%	99.80%	0.00%	0.00%	0.20%	0.00%	0.00%	0.00%	0.00%
	4	0.20%	0.00%	0.00%	99.80%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
	5	0.00%	0.40%	0.00%	0.00%	99.60%	0.00%	0.00%	0.00%	0.00%	0.00%
	6	0.00%	0.00%	0.00%	0.00%	0.00%	100.00%	0.00%	0.00%	0.00%	0.00%
	7	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	100.00%	0.00%	0.00%	0.00%
	8	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	100.00%	0.00%	0.00%
	9	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	100.00%	0.00%
	10	0.00%	0.00%	0.00%	0.00%	0.40%	0.00%	0.00%	0.00%	0.00%	99.60%
Min %C		98.20%									
Max %C		100.00%									
Overall %C		99.64%									

4.4 NRT Device Discrimination Analysis

Tables 4.7 and 4.8 were created to assess how well the RF air monitor classified and discriminated between devices in NRT after MDA_{Air} data was uploaded. Table 4.7 is a

raw CM consisting of the cumulative totals collected for each device. The highlighted diagonal entries represent the number of times each time each device was correctly classified and the non-diagonal values contains the the number of times the device was incorrectly classified (i.e. device 1 classified as device 2,3,etc N times). The cumulative sums at the right end of each row on Table 4.7 represent the total number of times each RZUSBstick was detected. The cumulative sums at the bottom of each column on Table 4.7 represent the total number of times a particular classification for a device was declared.

Table 4.7. Operational NRT Device Discrimination Raw CM Totals for $N_d = 10$ Devices

RF-DNA Totals	Classified Device Totals											Input Device Totals
	Device	1	2	3	4	5	6	7	8	9	10	
Input Device Totals	1	1574	0	0	335	0	0	0	0	0	0	1909
	2	12	1850	0	0	8	0	0	0	0	40	1910
	3	0	0	1910	0	0	0	0	1	0	0	1911
	4	0	0	0	1908	0	0	0	0	0	0	1908
	5	97	1	0	0	1814	0	0	0	0	0	1912
	6	0	0	0	0	0	1880	1	0	9	19	1909
	7	0	0	0	0	0	0	1885	0	25	0	1910
	8	0	0	0	0	0	0	0	1912	0	0	1912
	9	0	0	0	0	0	0	9	1	1899	0	1909
	10	0	0	0	0	0	27	0	0	1	1879	1907
Classified Device Totals		1683	1851	1910	2243	1822	1907	1895	1914	1934	1938	19097

The CM depicted on Table 4.8 was created with the N totals within each element in Table 4.7. Each element's percentage ($\%_{\text{Device,Class}}$) was calculated using,

$$\%_{\text{Device,Class}} = \frac{N_{\text{Device,Class}}}{\sum_{m=1}^{10} N_{\text{Device},m}}, \quad \text{Device} = \{1, \dots, 10\} \text{ and } \text{Class} = \{1, \dots, 10\}, \quad (4.5)$$

where the %C and %I of each device in Table 4.8 is

$$\%_{\text{Device,Class}} = \begin{cases} \%C, & \text{Device=Class} \\ \%I, & \text{otherwise} \end{cases}, \quad (4.6)$$

The operational tests classification results for the $N_d = 10$ RZUSBsticks varied between $82.45\% \leq \%C \leq 100.00\%$ with an overall $\%C = 96.93\%$. Table 4.7 was used to create the CM below on Table 4.8 along with a joint Probability Mass Function (PMF) Table in Appendix B for further analysis.

Table 4.8. Operational NRT Device Discrimination %C CM for $N_d = 10$ Devices

RF-DNA	Classified Device (%)										
	Device	1	2	3	4	5	6	7	8	9	10
Input Device (%)	1	82.45%	0.00%	0.00%	17.55%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
	2	0.63%	96.86%	0.00%	0.00%	0.42%	0.00%	0.00%	0.00%	0.00%	2.09%
	3	0.00%	0.00%	99.95%	0.00%	0.00%	0.00%	0.00%	0.05%	0.00%	0.00%
	4	0.00%	0.00%	0.00%	100.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
	5	5.07%	0.05%	0.00%	0.00%	94.87%	0.00%	0.00%	0.00%	0.00%	0.00%
	6	0.00%	0.00%	0.00%	0.00%	0.00%	98.48%	0.05%	0.00%	0.47%	1.00%
	7	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	98.69%	0.00%	1.31%	0.00%
	8	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	100.00%	0.00%	0.00%
	9	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.47%	0.05%	99.48%	0.00%
	10	0.00%	0.00%	0.00%	0.00%	0.00%	1.42%	0.00%	0.00%	0.05%	98.53%
Min %C		82.45%									
Max %C		100.00%									
Overall %C		96.93%									

V. Conclusion

This chapter provides a summary of the research activities conducted, research contributions, and recommendations for future research.

5.1 Results Summary

Wireless device usage is widespread and increasing as more consumers embrace Internet of Things (IoT) devices. ZigBee is a popular choice amongst consumers because of its low-cost, low-power, and low-bit rate communications are ideal for Wireless Sensor Network(s) (WSN). ZigBee Low-Rate Wireless Personal Area Network(s) (LR-WPAN) are extensively used in sensor networks, cyber-physical systems, and smart buildings [1]. With this use comes a heightened risk of unauthorized data access and modifications. Such problems occur when rogue users or devices bypass security measures by spoofing bit-level credentials. Prior research has shown fingerprinting to be effective as an additional form of Physical Layer (PHY) security against malicious attacks [6, 8, 12, 13, 16]. While proven effective through MATLAB simulation-based classification, this research aims to examine the effectiveness of fingerprinting and device discrimination in Near Real-Time (NRT) using an Radio Frequency (RF) air monitor employing an Universal Software Radio Peripheral(s) (USRP) Software-Defined Radio(s) (SDR) and open-source software.

5.2 Research Contribution

An RF air monitor for fingerprinting and discriminating devices in NRT is feasible and effective using trained Multiple Discriminant Analysis/Maximum Likelihood (MDA/ML) data. NRT Radio Frequency-Distinct Native Attribute (RF-DNA) Fingerprinting performance was first assessed by the air monitor's ability to capture and pro-

cess ZigBee bursts accurately and efficiently from the $N_d = 10$ RZUSBsticks while the systems was writing to the hard drive at a rate over $R_{\text{Store}} > 68$ MB/sec. There were some occasional buffer overflows that occurred because of the system resources allocated during the experiment. Despite the overflows, the air monitor acquired an Overall Packet Detection Percent ($\%D$) = 97.93% from the $N_{\text{Overall}} = 19,120$ transmitted ZigBee bursts. Each ZigBee contained a transmission period between $10.51 \text{ msec} \leq T_{\text{TX Burst}} \leq 13.07 \text{ msec}$ intervals. During the collection trial, the RF air monitor achieved an Overall Accurate Packet Reconstruction Percent ($\%R$) = 97.92% from the transmitted ZigBee data; 99.99% of the detected $N_{\text{Detected}} = 18,725$ ZigBee PHY Service Data Unit (PSDU) packets were properly reconstructed (i.e. passed Cyclic Redundancy Check (CRC)-16 verification); and the overall mean Absolute Reconstructed Packet Ratio Deviation ($\%A_{\Delta}$) was 0.0425% from the ideal reconstructed packet composition. The RF air monitor attained an Overall Fingerprinted Percent ($\%F$) = 97.48% from the overall transmitted Institute of Electrical and Electronics Engineers (IEEE) 802.15.4 Synchronization Headers (SHRs) and 99.65% of the $N_{\text{Accepted}} = 18,708$ accepted IEEE 802.15.4 SHRs bursts were fingerprinted. The IEEE 802.15.4 SHR bursts were reliably RF-DNA fingerprinted within $0.45 \text{ msec} \leq T_{\text{MDA/ML}} \leq 1.5 \text{ msec}$ after SHR burst detection. It's assumed that each device would be classified within the time same interval because non-trained data was used during the device discrimination process. Overall, the air monitor took an average $T_{\text{Collect}} = 25.651 \text{ sec}$ to detect and process $N_{\text{Bursts}} = 1,912$ ZigBee bursts transmitted from each RZUSBstick.

The focus of the second test was to assess the operational performance of the air monitor with trained MDA_{Air} data and the raw data storage options disabled. The same transmission conditions and performance metrics were used for the $N_d = 10$ devices as the initial experiment. The air monitor's overall detection performance increased to $\%D = 99.99\%$ from the $N_{\text{Overall}} = 19,120$ transmitted ZigBee bursts. Each

transmitted ZigBee bursts maintained the same transmission period between $10.51 \text{ msec} \leq T_{\text{TX Burst}} \leq 13.07 \text{ msec}$ as the initial collection trial. The RF air monitor's overall ZigBee packet reconstruction performance increased with $\%R = 99.96\%$ from the transmitted ZigBee data; 99.97% of the $N_{\text{Detected}} = 19,118$ ZigBee PSDU packets were properly reconstructed; and the overall mean $\%A_{\Delta}$ was 0.035% from the ideal reconstructed packet composition. The RF air monitors fingerprinting numbers also increased during the operational trials with $\%F = 99.88\%$ from the overall transmitted IEEE 802.15.4 SHRs and 99.97% of the $N_{\text{Accepted}} = 19,103$ accepted SHRs bursts were fingerprinted. The amount of time it took to fingerprint an IEEE 802.15.4 burst was not measured for the second performance assessment, but was assumed to be $0.45 \text{ msec} \leq T_{\text{TX Burst}} \leq 1.5 \text{ msec}$ or better. The average time the air monitor took to detect and process $N_{\text{Bursts}} = 1,912$ ZigBee bursts dropped to $T_{\text{Collect}} = 25.198 \text{ sec}$. The overall performance of the RF air monitor improved across every examined performance metric.

An estimative MDA/ML_{Est} performance model was developed with fingerprints created by adding Additive White Gaussian Noise (AWGN) to the $N_d = 10$ ZigBee device collections to simulate an environment with an Signal-to-Noise Ratio (SNR) range between $0 \text{ dB} \leq \text{SNR} \leq 40 \text{ dB}$ in 5 dB increments. The MDA/ML_{Air} model trained by the air monitor's fingerprints were compared against the estimative MDA/ML_{Est} model. The ZigBee RF-DNA fingerprints generated by the air monitor were collected at $\text{SNR} \approx 33.571 \text{ dB}$. The air monitors RF-DNA MDA/ML_{Air} model produced a Average Percent Correct Classification ($\%C$) = 99.64% that well exceeded the estimative MDA/ML_{Est} model range $98.76\% \leq \%C \leq 99.10\%$ at an $30 \text{ dB} \leq \text{SNR} \leq 35 \text{ dB}$. The trained MDA_{Air} data created from the NRT RF-DNA fingerprints were uploaded into the air monitor for the second test.

The air monitors NRT classification performance was assessed by creating a Confusion Matrix (CM) from the statistical data generated when the RF air monitor

was fingerprinting and discriminating devices. The classification results for the $N_d = 10$ RZUSBsticks varied between $82.45\% \leq \%C \leq 100.00\%$ with an overall $\%C = 96.93\%$. The non-simulated NRT classification differed slightly from the RF air monitors trained model with a $\%C$ Deviation ($\%C_{\Delta}$) = 2.71% lower than the MDA/ML_{Air} $\%C = 99.64\%$. Similarly, the NRT classification performance fell within the $20 \text{ dB} \leq \text{SNR} \leq 25 \text{ dB}$ range of the estimative MDA/ML_{Est} that varied between $95.76\% \leq \%C \leq 98.06\%$.

5.3 Future Research Recommendation

This research provides as a proof-of-concept demonstration using reasonably priced SDR and open-source tools that can be used to implement an RF air monitor for ZigBee devices. NRT RF-DNA Fingerprinting and device discrimination was successfully demonstrated using a B205mini-i and GNU Radio. The work completed provides a foundation for future research to:

1. Replicate the Experiment With Different Distinct Native Attribute (DNA) Fingerprint Features: Alternate DNA fingerprinting techniques could easily be implemented into the current GNU Radio air monitor. For example, Constellation Based-Distinct Native Attributes (CB-DNA) based Fingerprinting and discrimination has been shown to be more effective than RF-DNA in previous ZigBee research [8]. The CB-DNA experiments could be replicated over-the-air in NRT to see how the NRT results compare to past research.
2. Consider NRT Discrimination For Various IEEE Standards: This research was possible because an existing IEEE 802.15.4 module was available on GNU Radio. Various IEEE standards have successfully been deployed in GNU Radio. Most of these modules are openly available on GitHub and could potentially be transformed into air monitor prototypes.

3. Consider Autonomous Model Training: The RF air monitor's ability to discriminate between devices was entirely dependent on the MDA/ML model developed offline in MATLAB. Creating a model in MATLAB is not ideal for an NRT system. Properly formatting the GNU Radio fingerprints so that MDA/ML model can be created takes some time. Research towards the development of an intelligently trained autonomous MDA/ML model could potentially lead to the development of a truly "stand-alone" air monitor.

Appendix A. Device Collection Figures

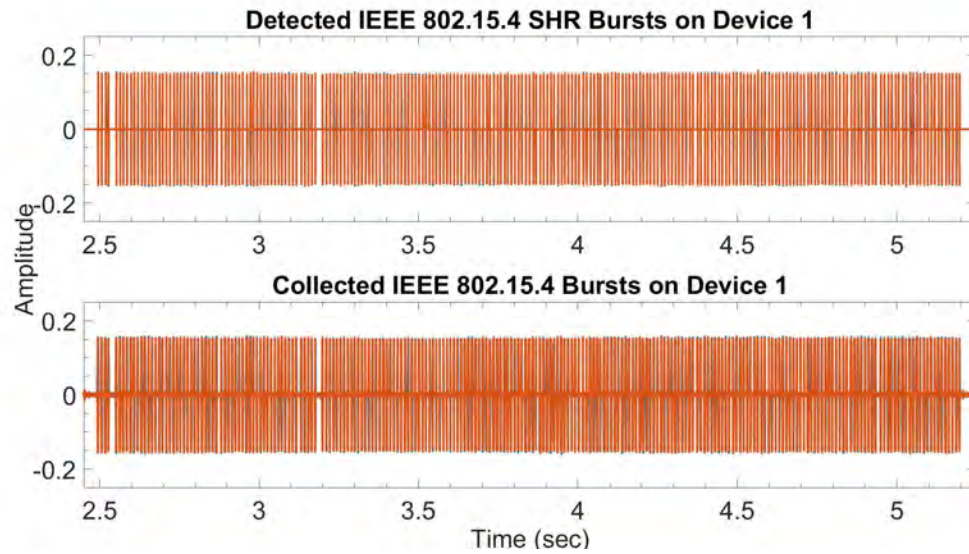


Figure A-1. I/Q Plot on Device 1 of $N_{\text{Bursts}}=239$ bursts from Libpcap File

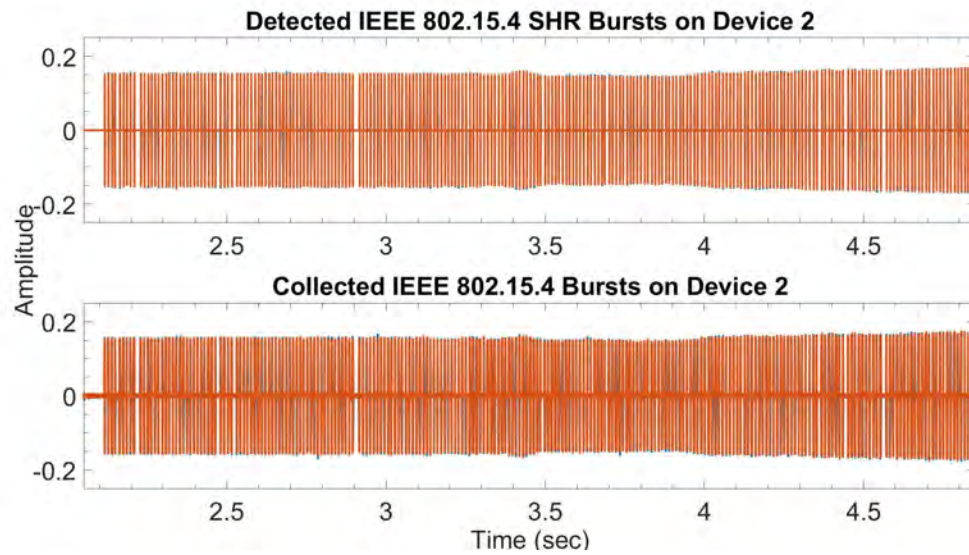


Figure A-2. I/Q Plot on Device 2 of $N_{\text{Bursts}}=239$ bursts from Libpcap File

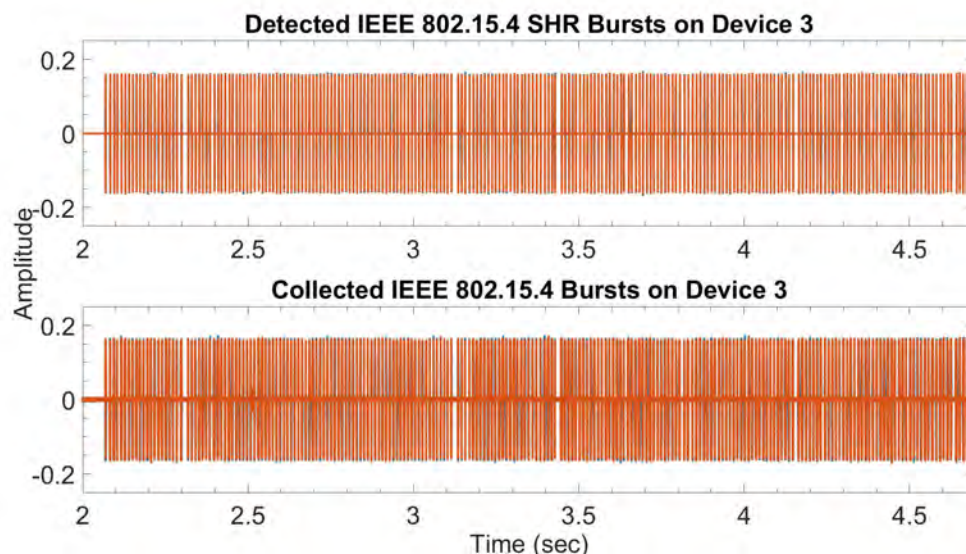


Figure A-3. I/Q Plot on Device 3 of $N_{\text{Bursts}}=239$ bursts from Libpcap File

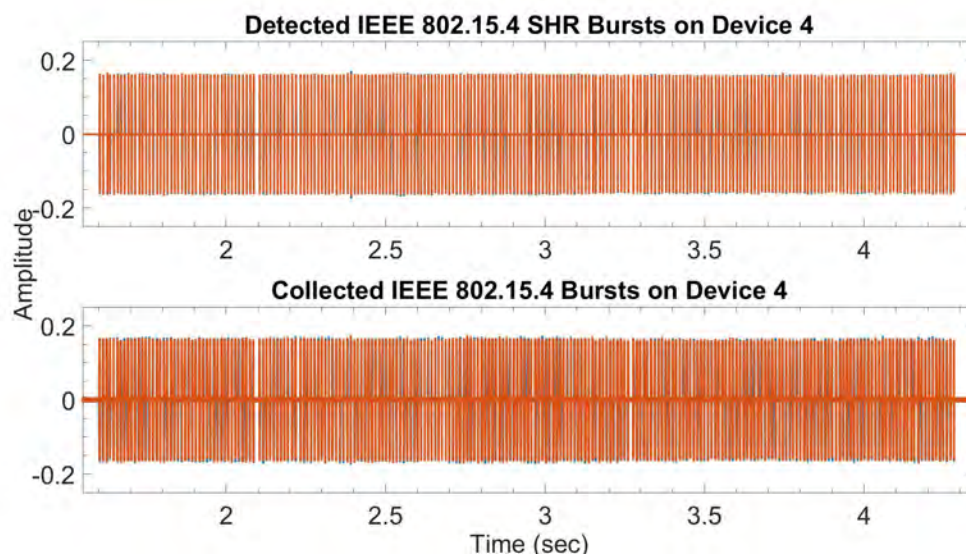


Figure A-4. I/Q Plot on Device 4 of $N_{\text{Bursts}}=239$ bursts from Libpcap File

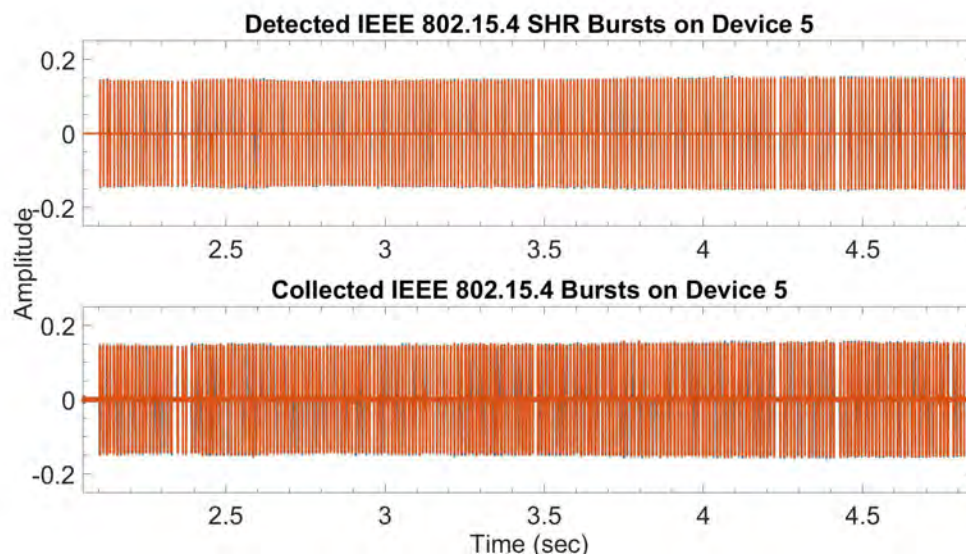


Figure A-5. I/Q Plot on Device 5 of $N_{\text{Bursts}}=239$ bursts from Libpcap File

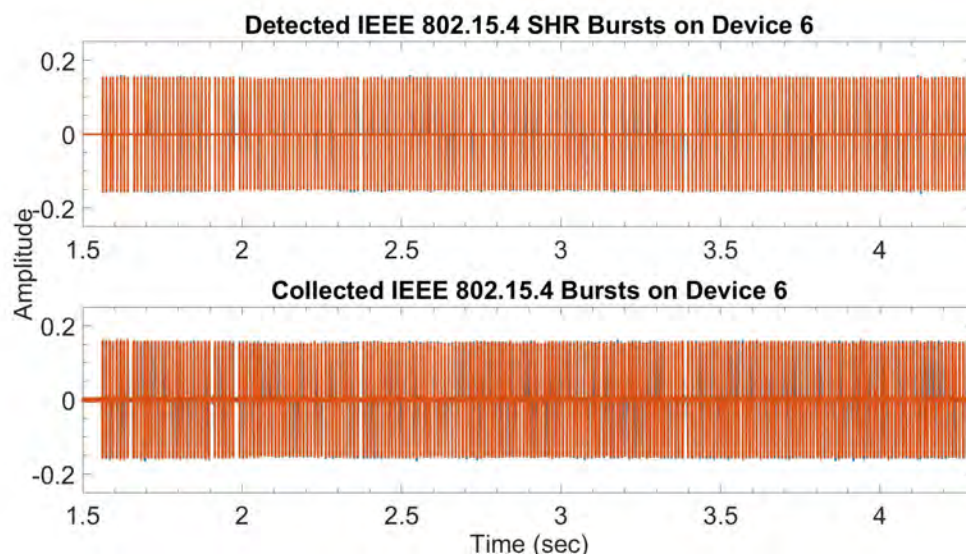


Figure A-6. I/Q Plot on Device 6 of $N_{\text{Bursts}}=239$ bursts from Libpcap File

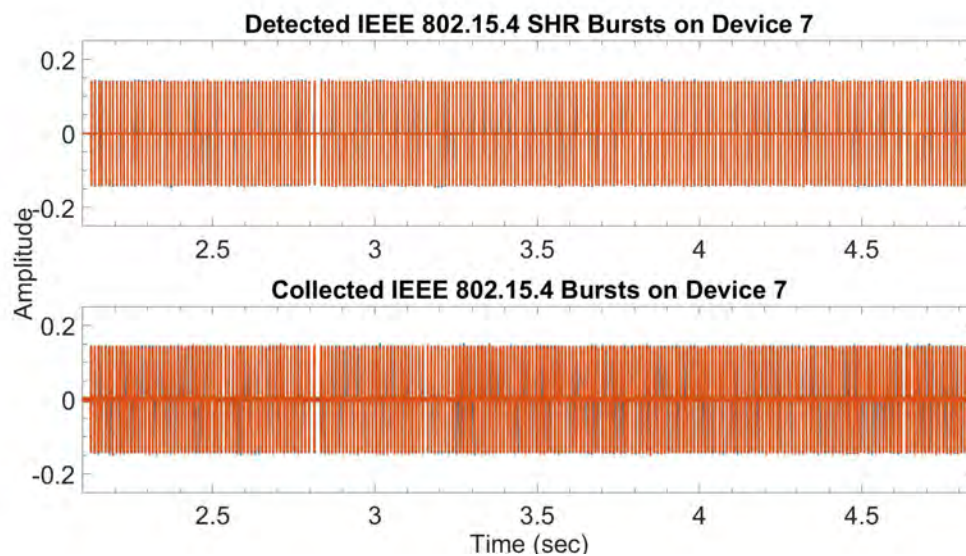


Figure A-7. I/Q Plot on Device 7 of $N_{\text{Bursts}}=239$ bursts from Libpcap File

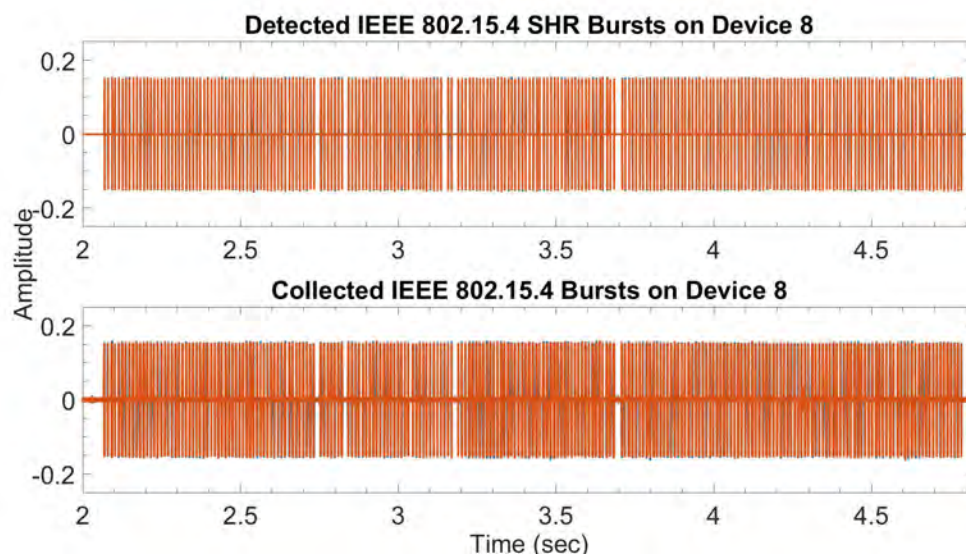


Figure A-8. I/Q Plot on Device 8 of $N_{\text{Bursts}}=239$ bursts from Libpcap File

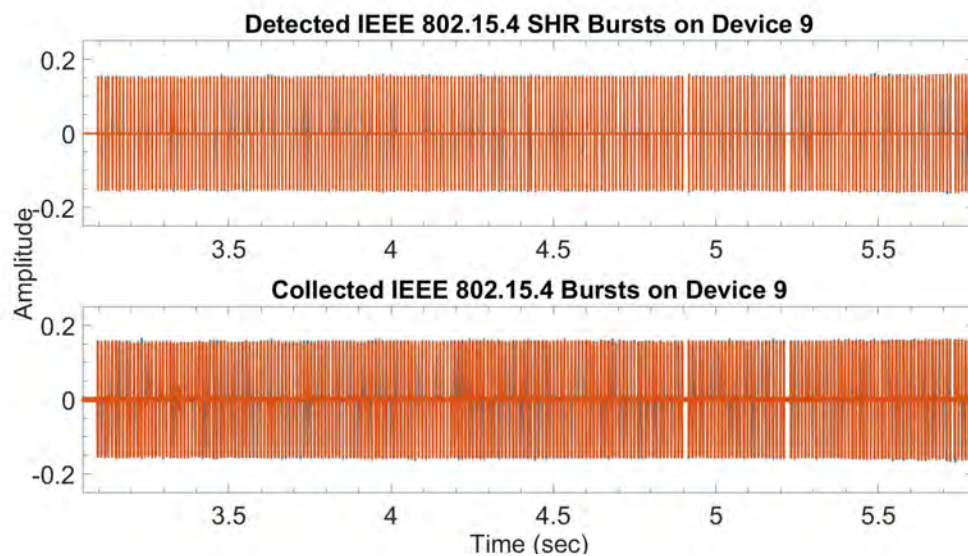


Figure A-9. I/Q Plot on Device 9 of $N_{\text{Bursts}}=239$ bursts from Libpcap File

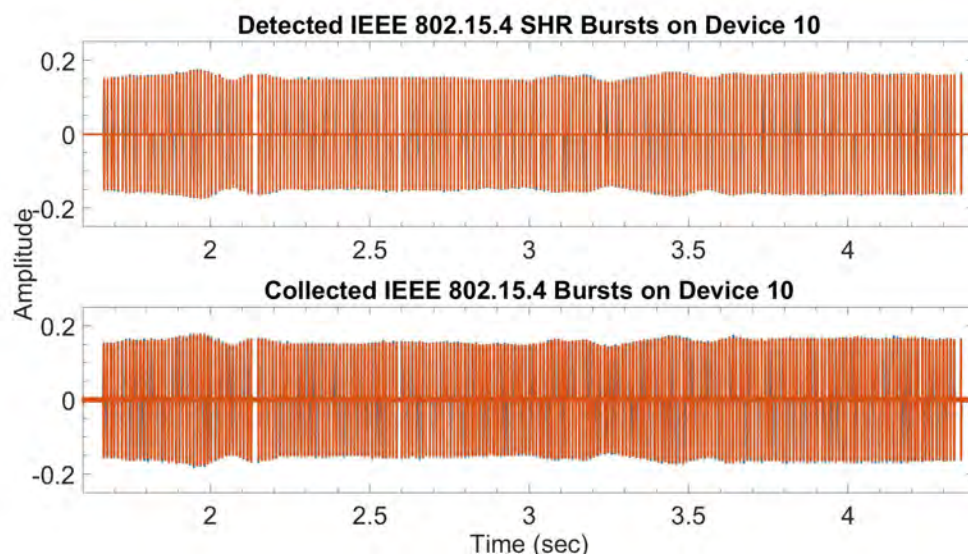


Figure A-10. I/Q Plot on Device 10 of $N_{\text{Bursts}}=239$ bursts from Libpcap File

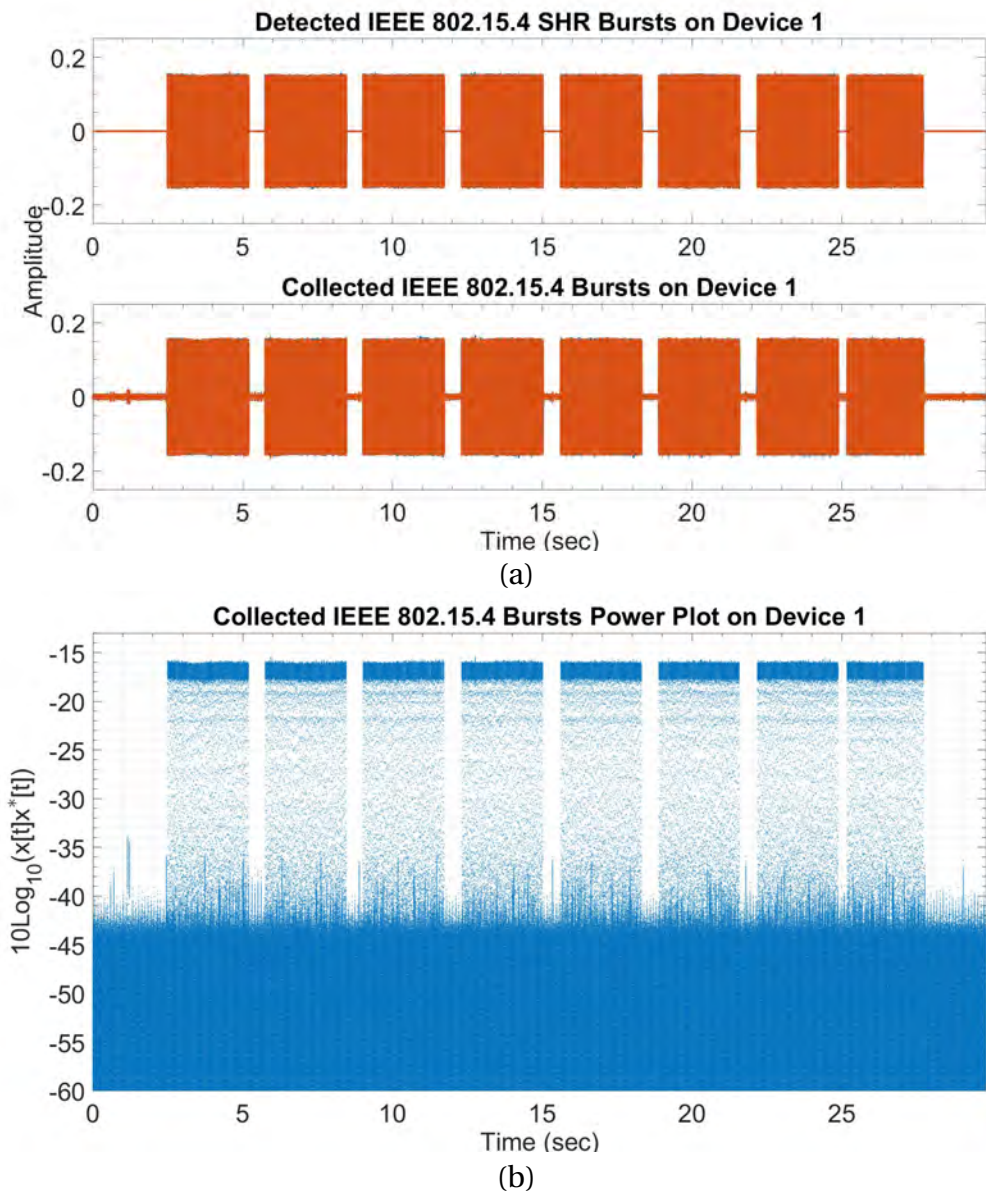


Figure A-11. (a) I/Q and (b) Power Plot on Device 1 of $N_{\text{Bursts}}=1,912$ bursts from Libpcap File Replayed 8-Times

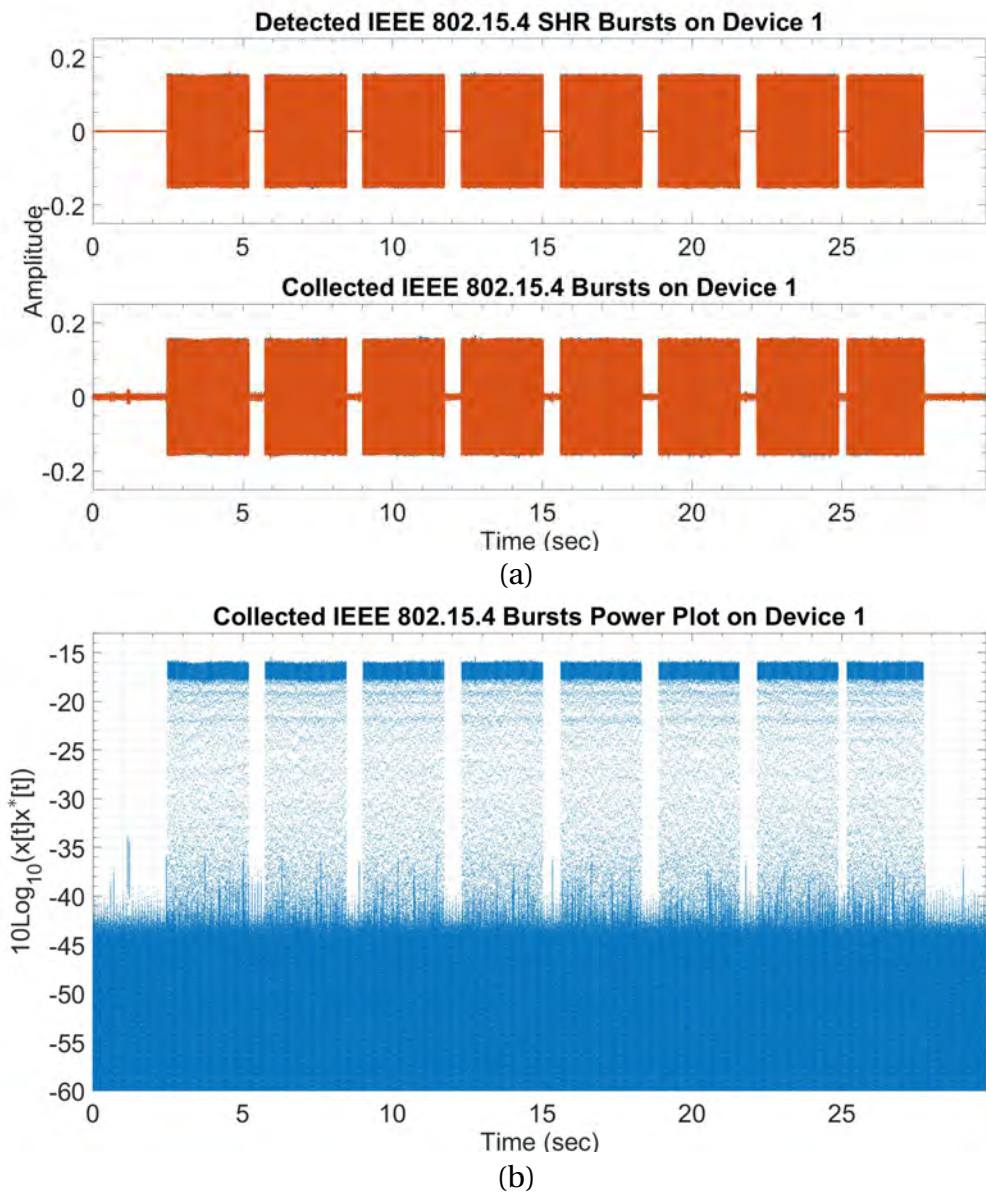


Figure A-12. (a) I/Q and (b) Power Plot on Device 1 of $N_{\text{Bursts}}=1,912$ bursts from Libpcap File Replayed 8-Times

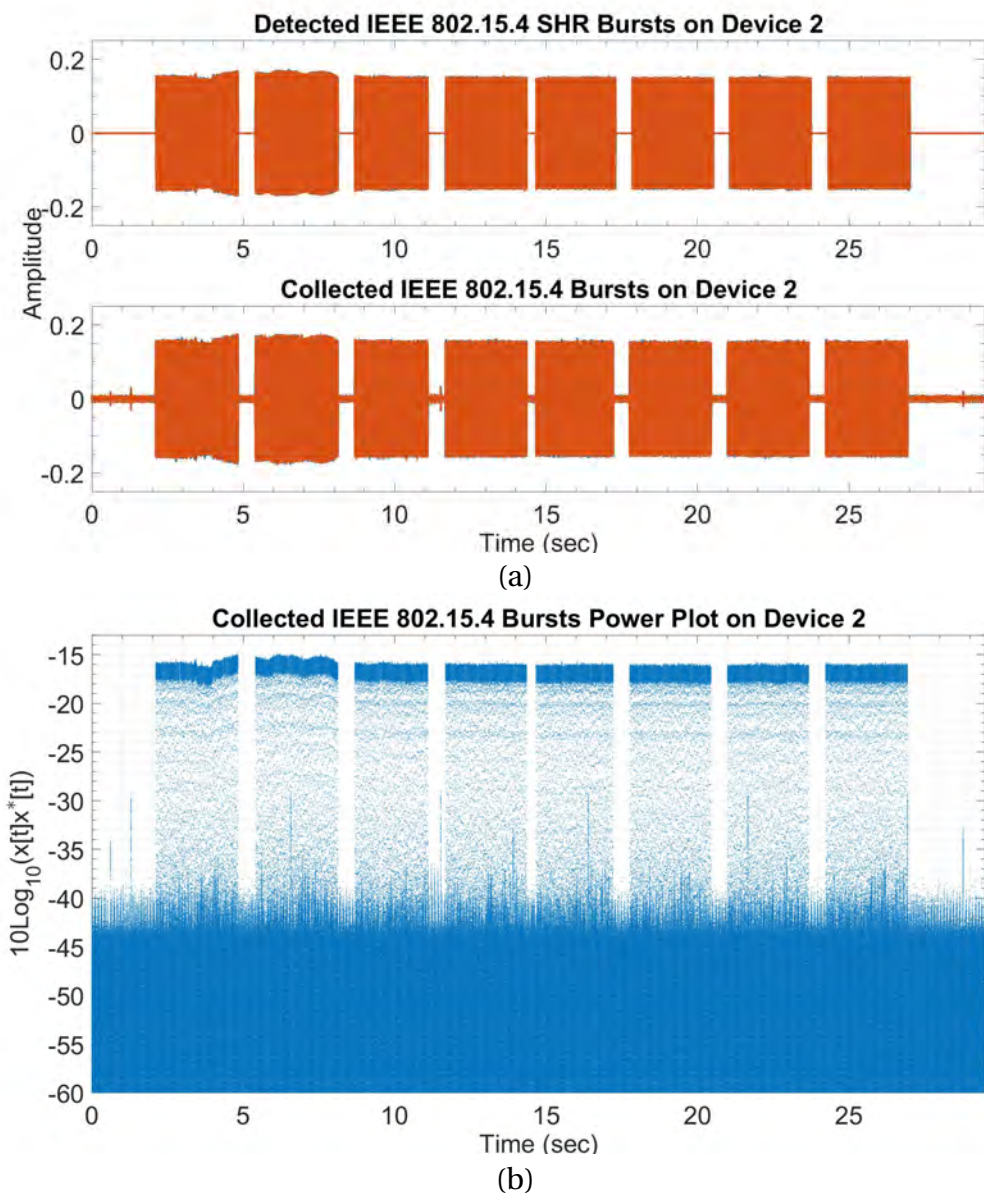


Figure A-13. (a) I/Q and (b) Power Plot on Device 2 of $N_{\text{Bursts}} = 1,912$ bursts from Libpcap File Replayed 8-Times

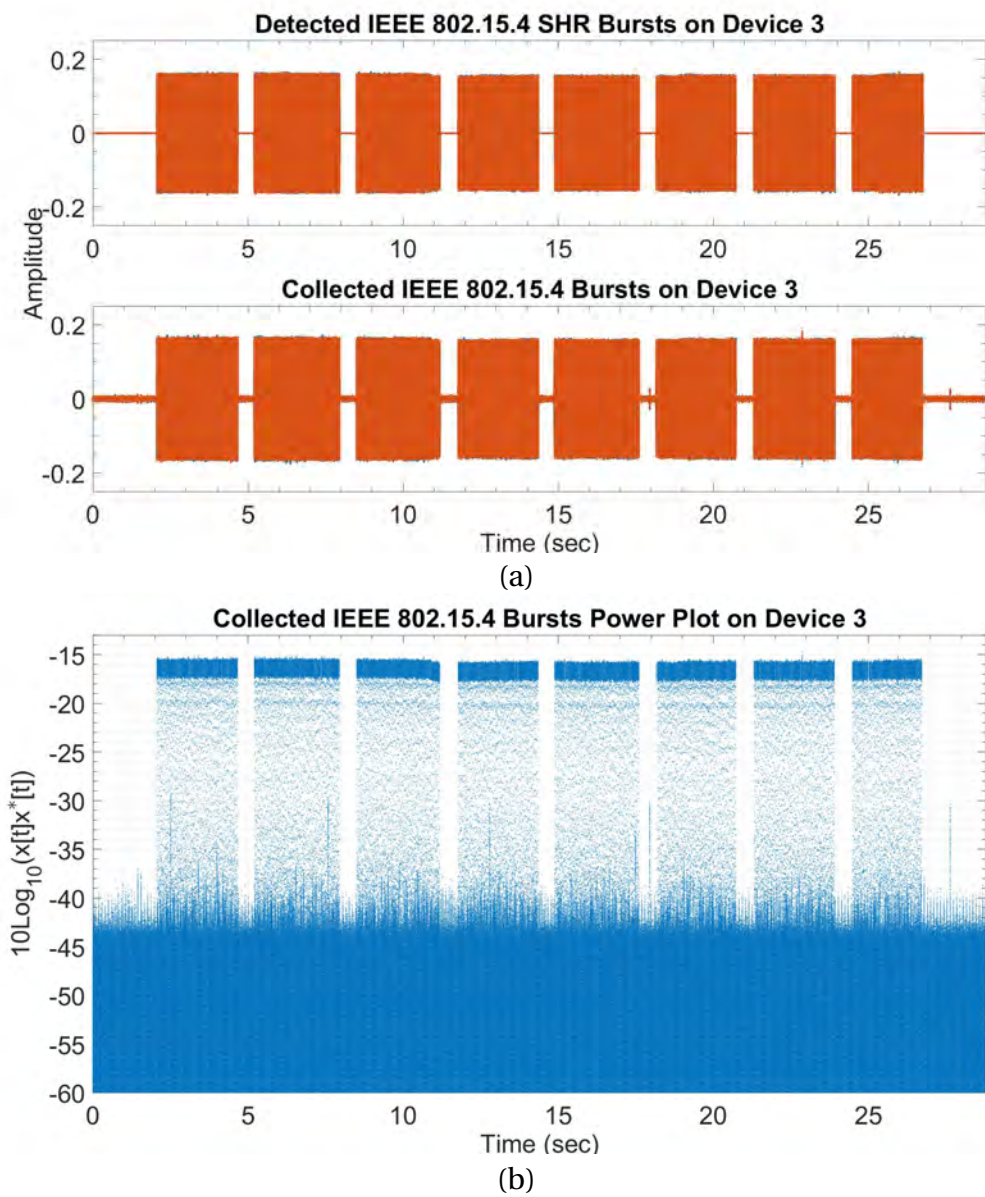


Figure A-14. (a) I/Q and (b) Power Plot on Device 3 of $N_{\text{Bursts}} = 1,912$ bursts from Libpcap File Replayed 8-Times

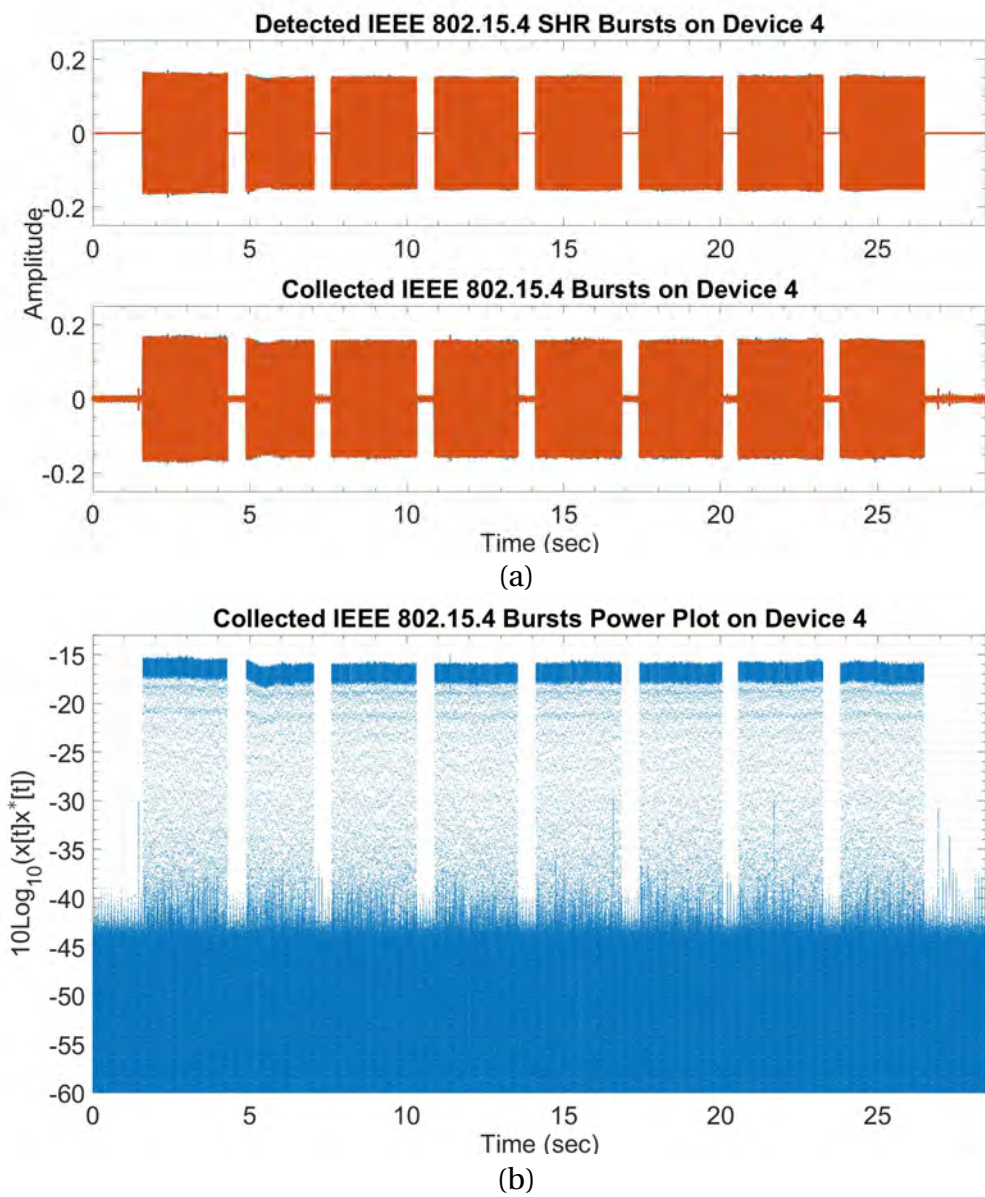


Figure A-15. (a) I/Q and (b) Power Plot on Device 4 of $N_{\text{Bursts}}=1,912$ bursts from Libpcap File Replayed 8-Times

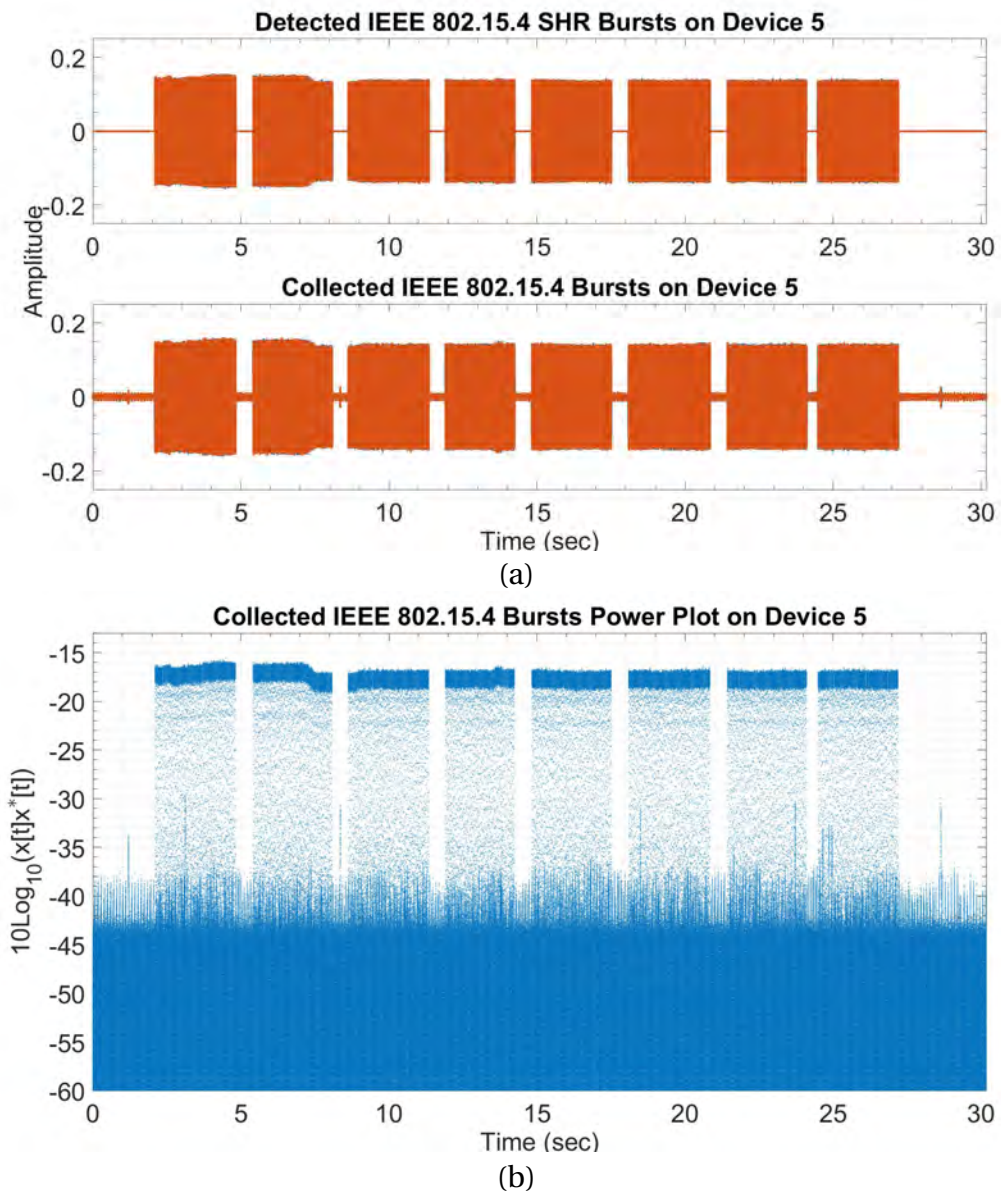


Figure A-16. (a) I/Q and (b) Power Plot on Device 5 of $N_{\text{Bursts}}=1,912$ bursts from Libpcap File Replayed 8-Times

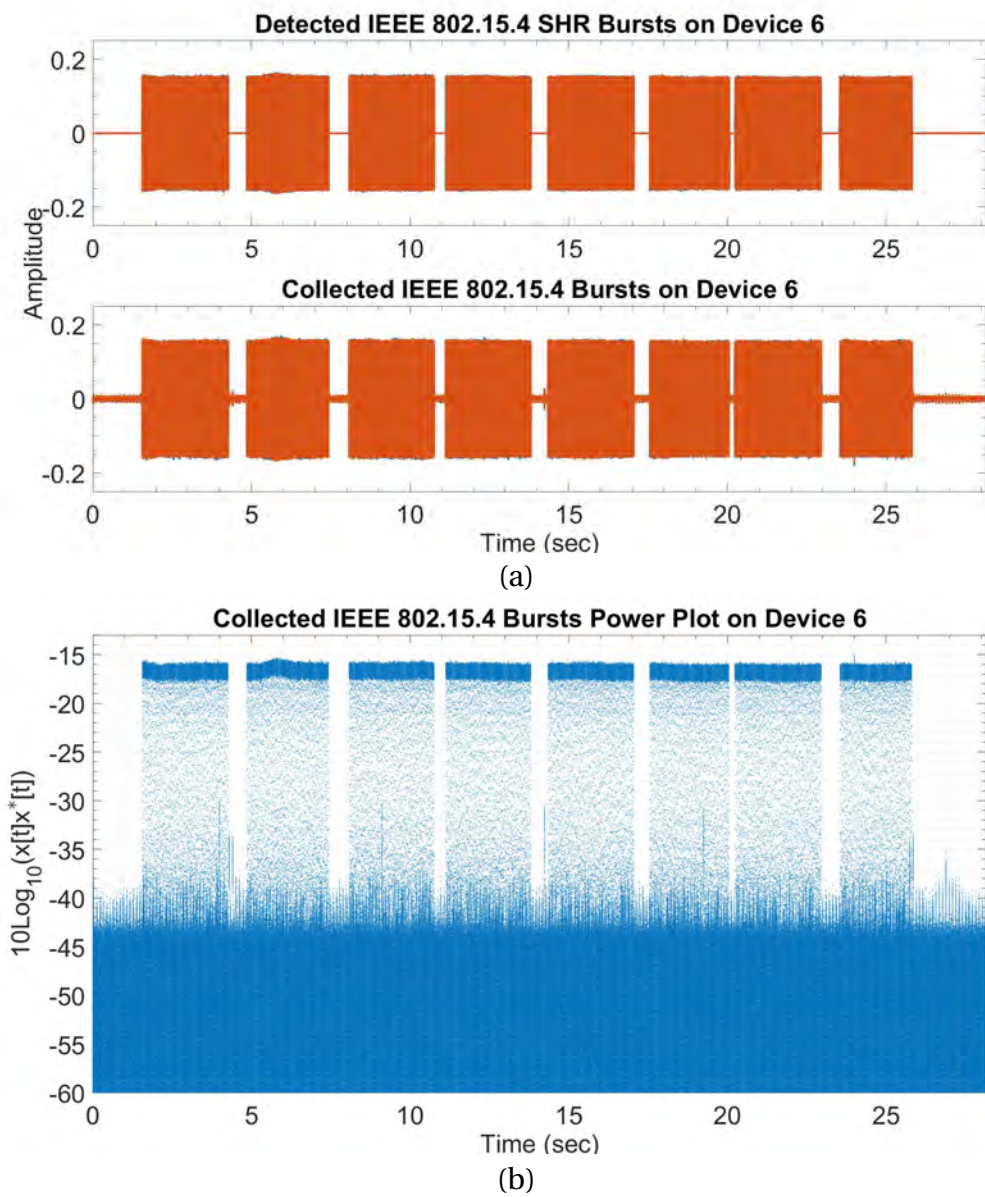


Figure A-17. (a) I/Q and (b) Power Plot on Device 6 of $N_{\text{Bursts}}=1,912$ bursts from Libpcap File Replayed 8-Times

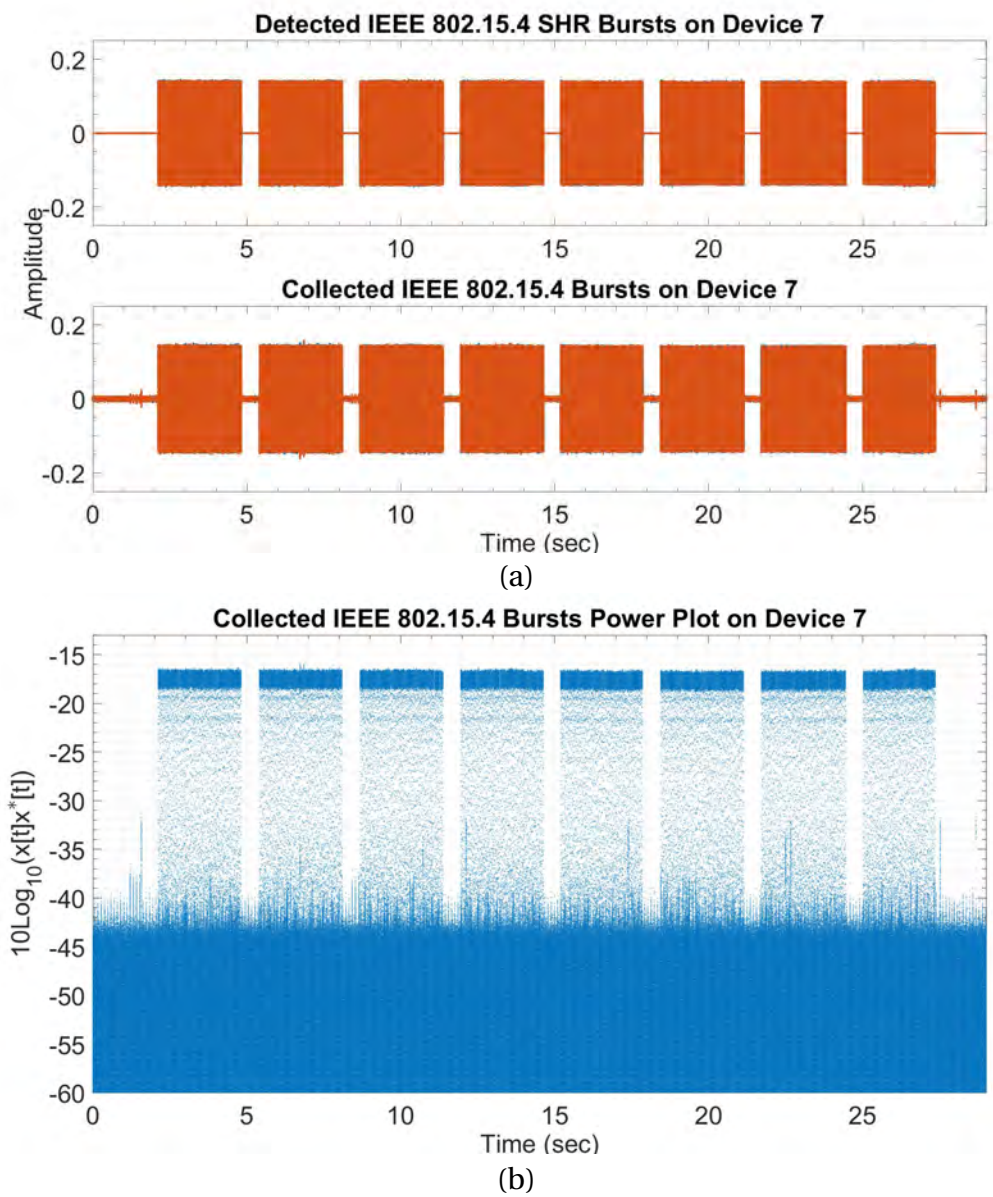


Figure A-18. (a) I/Q and (b) Power Plot on Device 7 of $N_{\text{Bursts}} = 1,912$ bursts from Libpcap File Replayed 8-Times

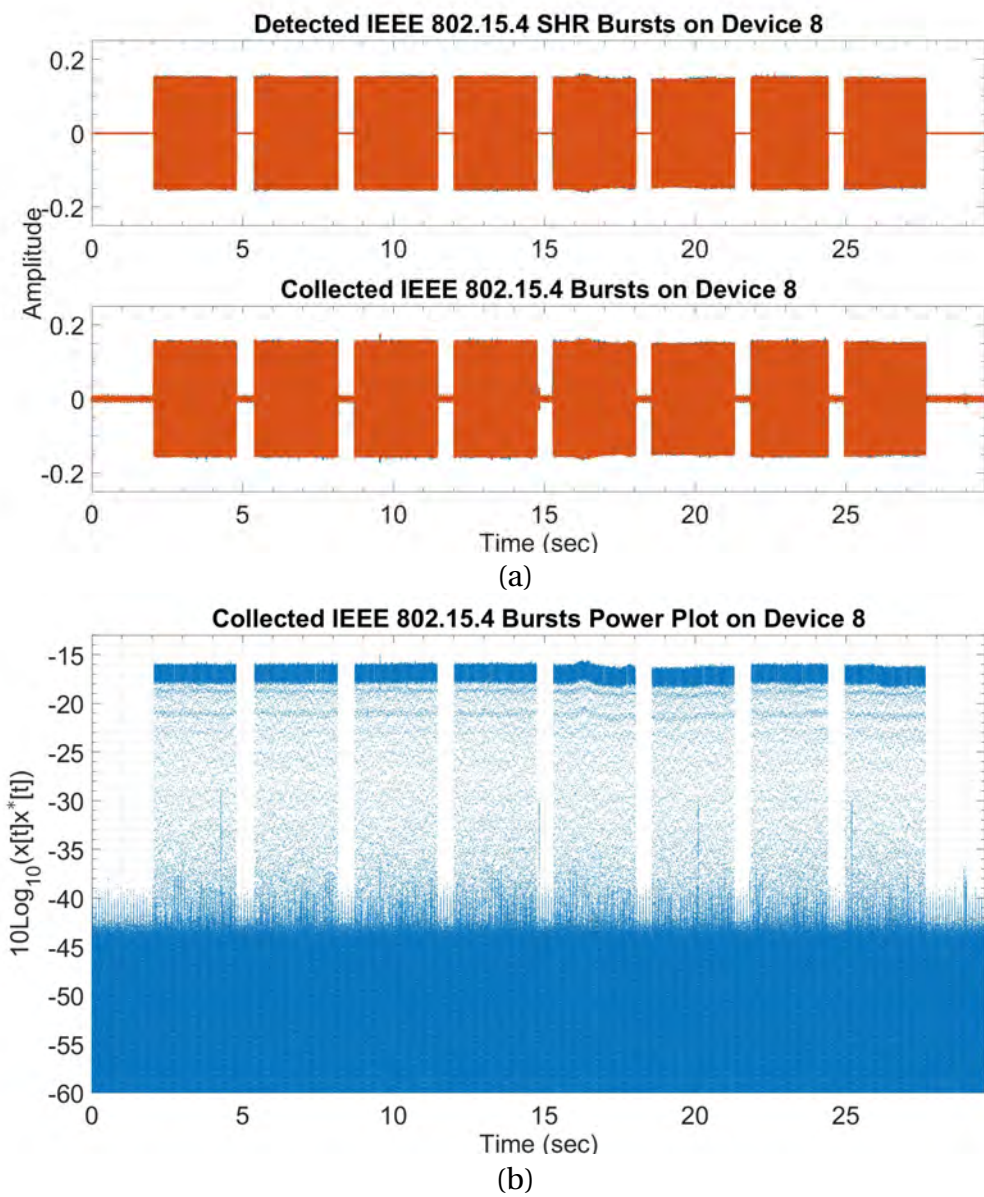


Figure A-19. (a) I/Q and (b) Power Plot on Device 8 of $N_{\text{Bursts}} = 1,912$ bursts from Libpcap File Replayed 8-Times

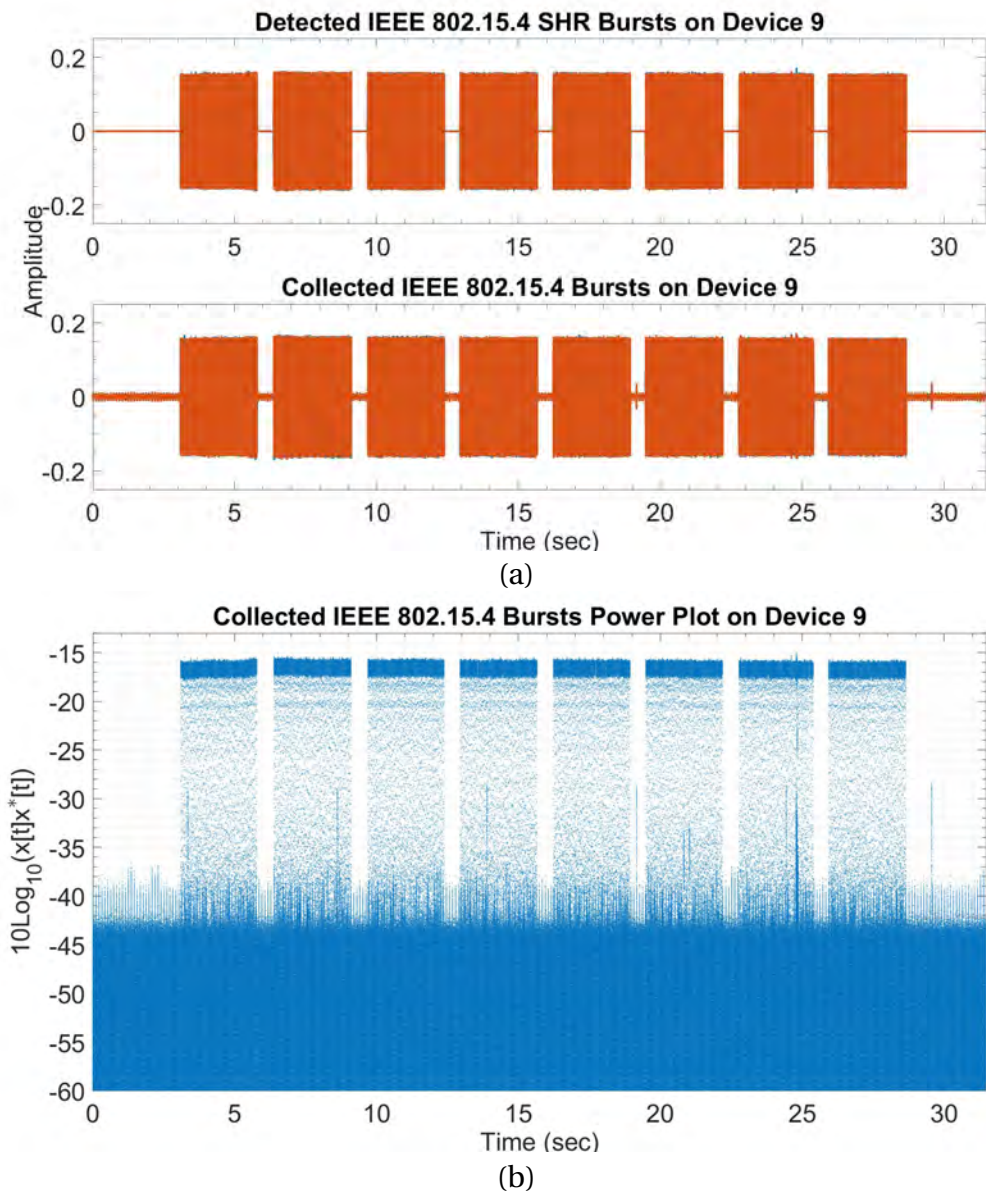


Figure A-20. (a) I/Q and (b) Power Plot on Device 9 of $N_{\text{Bursts}}=1,912$ bursts from Libpcap File Replayed 8-Times

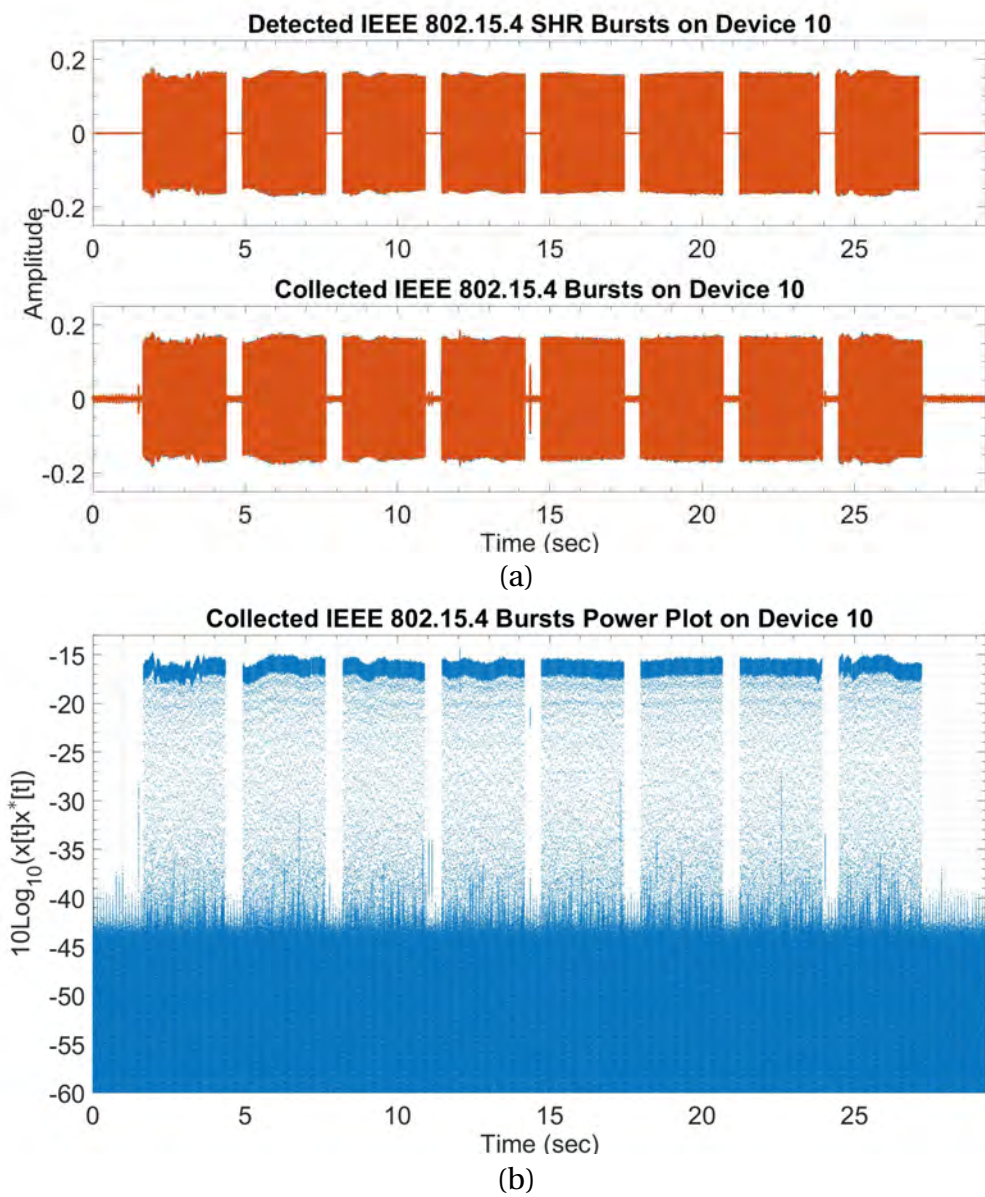


Figure A-21. (a) I/Q and (b) Power Plot on Device 10 of $N_{\text{Bursts}} = 1,912$ bursts from Libpcap File Replayed 8-Times

Appendix B. Device Collection Tables

Table 1. Collection Performance Tables for Devices 1 to 4

Device: 1											
Time Span (sec)			25.7								
			Detected IEEE 802.15.4 Libpcap Data								
Total Packets Transmitted			100.00%	1912	Packet Lengths			Packet Count	Average Val (B)	Min Val(B)	Max Val (B)
Preamble Sink Detected Bursts	Total Packets Detected		99.22%	1897 <th>Total Packets</th> <td>100.00%</td> <td>1897</td> <td>58.54</td> <td>10</td> <td>90</td>		Total Packets	100.00%	1897	58.54	10	90
	Short Packets		0.00%	0		0-19	3.32%	63	12.32	10	18
	Bad CRC Packets		0.05%	1		20-39	2.53%	48	26.67	21	28
	Good CRC Packets		99.16%	1896		40-79	69.11%	1311	51.87	45	73
	Preambles Accepted		99.01%	1893		80-159	25.04%	475	86.28	80	90
	Preambles Rejected		0.21%	4							
	Total Processed Fingerprints		98.80%	1889		% Preamble's Fingerprinted		99.79%			

Device: 2										
Time Span (sec)				25.548						
				Detected IEEE 802.15.4 Libpcap Data						
Total Packets Transmitted		100.00%	1912	Packet Lengths		Packet Count	Average Val (B)	Min Val(B)	Max Val (B)	
Preamble Sink Detected Bursts	Total Packets Detected	97.96%	1873 <th>Total Packets</th> <td>100.00%</td> <td>1873</td> <td>58.76</td> <td>10</td> <td>90</td>		Total Packets	100.00%	1873	58.76	10	90
	Short Packets	0.00%	0		0-19	3.20%	60	12.23	10	18
	Bad CRC Packets	0.00%	0		20-39	2.24%	42	26.67	21	28
	Good CRC Packets	97.96%	1873		40-79	69.25%	1297	51.89	45	73
	Preambles Accepted	97.80%	1870		80-159	25.31%	474	86.27	80	90
	Preambles Rejected	0.16%	3							
Total Processed Fingerprints		97.59%	1866 <th colspan="2">% Preamble's Fingerprinted</th> <td colspan="4">99.79%</td>	% Preamble's Fingerprinted		99.79%				

Device: 3											
Time Span (sec)				25.89							
				Detected IEEE 802.15.4 Libpcap Data							
		Total Packets Transmitted	100.00%	1912	Packet Lengths			Packet Count	Average Val (B)	Min Val(B)	Max Val (B)
Preamble Sink Detected Bursts	Total Packets Detected		95.19%	1820		Total Packets	100.00%	1820	58.85	10	90
	Short Packets		0.00%	0		0-19	3.24%	59	12.24	10	18
	Bad CRC Packets		0.00%	0		20-39	2.47%	45	26.73	21	28
	Good CRC Packets		95.19%	1820		40-79	68.63%	1249	51.95	45	73
	Preambles Accepted		95.19%	1820		80-159	25.66%	467	86.28	80	90
	Preambles Rejected		0.00%	0							
Total Processed Fingerprints		94.19%	1801	% Preamble's Fingerprinted		98.96%					

Device : 4												
Time Span (sec)			25.758									
Total Packets Transmitted			100.00%	1912	Packet Lengths			Packet Count	Average Val (B)	Min Val(B)	Max Val (B)	
Preamble Sink Detected Bursts	Total Packets Detected		96.13%	1838 <th>Total Packets</th> <td>100.00%</td> <td>1838</td> <td>58.47</td> <td>10</td> <td>90</td>		Total Packets	100.00%	1838	58.47	10	90	
	Short Packets		0.00%	0		0-19		3.37%	62	12.16	10	18
	Bad CRC Packets		0.00%	0		20-39		2.50%	46	26.78	21	28
	Good CRC Packets		96.13%	1838		40-79		69.15%	1271	51.8	45	73
	Preambles Accepted		96.03%	1836		80-159		24.97%	459	86.37	80	90
	Preambles Rejected		0.10%	2								
Total Processed Fingerprints			95.19%	1820	% Preamble's Fingerprinted		99.13%					

Table 2. Collection Performance Tables for Devices 5 to 8

Device: 5												
Time Span (sec)			25.71									
			Detected IEEE 802.15.4 Libpcap Data									
Total Packets Transmitted			100.00%	1912	Packet Lengths			Packet Count	Average Val (B)	Min Val(B)	Max Val (B)	
Preamble Sink Detected Bursts	Total Packets Detected		97.75%	1869 <th>Total Packets</th> <td>100.00%</td> <td>1869</td> <td>58.51</td> <td>1</td> <td>90</td>		Total Packets	100.00%	1869	58.51	1	90	
	Short Packets		0.05%	1		0-19		3.42%	64	11.98	1	18
	Bad CRC Packets		0.00%	0		20-39		2.57%	48	26.67	21	28
	Good CRC Packets		97.70%	1868		40-79		68.81%	1286	51.84	45	73
	Preambles Accepted		97.65%	1867		80-159		25.20%	471	86.27	80	90
	Preambles Rejected		0.10%	2								
	Total Processed Fingerprints		97.44%	1863		% Preamble's Fingerprinted		99.79%				

Device: 6											
Time Span (sec)			25.638								
			Detected IEEE 802.15.4 Libpcap Data								
Total Packets Transmitted			100.00%	1912	Packet Lengths			Packet Count	Average Val (B)	Min Val(B)	Max Val (B)
Preamble Sink Detected Bursts	Total Packets Detected		96.29%	1841		Total Packets	100.00%	1841	58.57	10	90
	Short Packets		0.00%	0		0-19	3.31%	61	12.26	10	18
	Bad CRC Packets		0.00%	0		20-39	2.61%	48	26.67	21	28
	Good CRC Packets		96.29%	1841		40-79	68.93%	1269	51.91	45	73
	Preambles Accepted		96.29%	1841		80-159	25.15%	463	86.22	80	90
	Preambles Rejected		0.00%	0							
Total Processed Fingerprints			95.50%	1826		% Preamble's Fingerprinted		99.19%			

Device: 7												
Time Span (sec)			25.665									
			Detected IEEE 802.15.4 Libpcap Data									
Preamble Sink Detected Bursts	Total Packets Transmitted		100.00%	1912	Packet Lengths			Packet Count	Average Val (B)	Min Val(B)	Max Val (B)	
	Total Packets Detected		98.27%	1879 <th>Total Packets</th> <td>100.00%</td> <td>1879</td> <td>58.46</td> <td>10</td> <td>90</td>		Total Packets	100.00%	1879	58.46	10	90	
	Short Packets		0.00%	0		0-19		3.35%	63	12.25	10	18
	Bad CRC Packets		0.00%	0		20-39		2.55%	48	26.67	21	28
	Good CRC Packets		98.27%	1879		40-79		69.19%	1300	51.89	45	73
	Preambles Accepted		98.27%	1879		80-159		24.91%	468	86.18	80	90
	Preambles Rejected		0.00%	0								
	Total Processed Fingerprints		98.01%	1874 <th colspan="2">% Preamble's Fingerprinted</th> <td colspan="5">99.73%</td>		% Preamble's Fingerprinted		99.73%				

Device: 8												
Time Span (sec)				25.755								
				Detected IEEE 802.15.4 Libpcap Data								
Preamble Sink Detected Bursts	Total Packets Transmitted		100.00%	1912	Packet Lengths			Packet Count	Average Val (B)	Min Val(B)	Max Val (B)	
	Total Packets Detected		99.16%	1896 <th>Total Packets</th> <td>100.00%</td> <td>1896</td> <td>58.52</td> <td>10</td> <td>90</td>		Total Packets	100.00%	1896	58.52	10	90	
	Short Packets		0.00%	0 <th colspan="2">0-19</th> <td>3.38%</td> <td>64</td> <td>12.25</td> <td>10</td> <td>18</td>		0-19		3.38%	64	12.25	10	18
	Bad CRC Packets		0.00%	0 <th colspan="2">20-39</th> <td>2.53%</td> <td>48</td> <td>26.67</td> <td>21</td> <td>28</td>		20-39		2.53%	48	26.67	21	28
	Good CRC Packets		99.16%	1896		40-79		69.04%	1309	51.86	45	73
	Preambles Accepted		98.95%	1892		80-159		25.05%	475	86.33	80	90
	Preambles Rejected		0.21%	4 <th colspan="7"></th>								
	Total Processed Fingerprints		98.90%	1891 <th colspan="2">% Preamble's Fingerprinted</th> <td colspan="6">99.95%</td>		% Preamble's Fingerprinted		99.95%				

Table 3. Collection Performance Tables for Devices 9 and 10

Device: 9												
Time Span (sec)			25.712									
			Detected IEEE 802.15.4 Libpcap Data									
Preamble Sink Detected Bursts	Total Packets Transmitted		100.00%	1912	Packet Lengths			Packet Count	Average Val (B)	Min Val(B)	Max Val (B)	
	Total Packets Detected		99.48%	1902 <th>Total Packets</th> <td>100.00%</td> <td>1902</td> <td>58.5</td> <td>10</td> <td>90</td>		Total Packets	100.00%	1902	58.5	10	90	
	Short Packets		0.00%	0		0-19		3.36%	64	12.25	10	18
	Bad CRC Packets		0.00%	0		20-39		2.52%	48	26.67	21	28
	Good CRC Packets		99.48%	1902		40-79		69.14%	1315	51.89	45	73
	Preambles Accepted		99.42%	1901		80-159		24.97%	475	86.23	80	90
	Preambles Rejected		0.05%	1								
Total Processed Fingerprints			99.37%	1900 <th colspan="2">% Preamble's Fingerprinted</th> <td colspan="4">99.95%</td>	% Preamble's Fingerprinted		99.95%					

Device: 10												
Time Span (sec)			25.534									
			Detected IEEE 802.15.4 Libpcap Data									
Preamble Sink Detected Bursts	Total Packets Transmitted		100.00%	1912	Packet Lengths			Packet Count	Average Val (B)	Min Val(B)	Max Val (B)	
	Total Packets Detected		99.90%	1910		Total Packets	100.00%	1910	58.54	10	90	
	Short Packets		0.00%	0		0-19		3.35%	64	12.25	10	18
	Bad CRC Packets		0.00%	0		20-39		2.51%	48	26.67	21	28
	Good CRC Packets		99.90%	1910		40-79		69.06%	1319	51.87	45	73
	Preambles Accepted		99.84%	1909		80-159		25.08%	479	86.26	80	90
	Preambles Rejected		0.05%	1								
Total Processed Fingerprints			99.84%	1909 <th colspan="2">% Preamble's Fingerprinted</th> <td colspan="4">100.00%</td>	% Preamble's Fingerprinted		100.00%					

Table 4. Operational Performance Tables for Devices 1 to 4

Device: 1										
Time Span (sec)		25.166								
		Detected IEEE 802.15.4 Libpcap Data								
Preamble Sink Detected Bursts	Total Packets Transmitted	100.00%	1912	Packet Lengths			Packet Count	Average Val (B)	Min Val (B)	Max Val (B)
	Total Packets Detected	99.95%	1911		Total Packets	100.00%	1911	58.52	10	90
	Short Packets	0.00%	0		0-19	3.40%	65	12.32	10	18
	Bad CRC Packets	0.05%	1		20-39	2.51%	48	26.67	21	28
	Good CRC Packets	99.90%	1910		40-79	69.02%	1319	51.87	45	73
	Preambles Accepted	99.84%	1909		80-159	25.07%	479	86.27	80	90
	Preambles Rejected	0.10%	2							
	Total Processed Fingerprints		99.84%		1909	% Preamble's Fingerprinted	100.00%			

Device: 2											
Time Span (sec)				25.309							
				Detected IEEE 802.15.4 Libpcap Data							
Preamble Sink Detected Bursts	Total Packets Transmitted		100.00%	1912	Packet Lengths			Packet Count	Average Val (B)	Min Val (B)	Max Val (B)
	Total Packets Detected		100.05%	1913 <th>Total Packets</th> <td>100.00%</td> <td>1913</td> <td>58.52</td> <td>1</td> <td>90</td>		Total Packets	100.00%	1913	58.52	1	90
	Short Packets		0.05%	1		0-19	3.40%	65	12.08	1	18
	Bad CRC Packets		0.00%	0		20-39	2.51%	48	26.67	21	28
	Good CRC Packets		100.00%	1912		40-79	69.00%	1320	51.87	45	73
	Preambles Accepted		99.90%	1910		80-159	25.09%	480	86.27	80	90
	Preambles Rejected		0.16%	3							
	Total Processed Fingerprints		99.90%	1910 <th>% Preamble's Fingerprinted</th> <td colspan="6">100.00%</td>		% Preamble's Fingerprinted	100.00%				

Device: 3												
Time Span (sec)		25.215										
				Detected IEEE 802.15.4 Libpcap Data								
Preamble Sink Detected Bursts	Total Packets Transmitted		100.00%	1912	Packet Lengths			Packet Count	Average Val (B)	Min Val (B)	Max Val (B)	
	Total Packets Detected		100.00%	1912 <th>Total Packets</th> <td>100.00%</td> <td>1912</td> <td>58.52</td> <td>1</td> <td>90</td>		Total Packets	100.00%	1912	58.52	1	90	
	Short Packets		0.05%	1		0-19	3.40%	65	12.08	1	18	
	Bad CRC Packets		0.00%	0		20-39	2.51%	48	26.67	21	28	
	Good CRC Packets		99.95%	1911		40-79	68.99%	1319	51.87	45	73	
	Preambles Accepted		99.95%	1911		80-159	25.10%	480	86.27	80	90	
	Preambles Rejected		0.05%	1								
	Total Processed Fingerprints		99.95%	1911 <th colspan="2">% Preamble's Fingerprinted</th> <td colspan="5">100.00%</td>		% Preamble's Fingerprinted		100.00%				

Device: 4											
Time Span (sec)				25.108							
				Detected IEEE 802.15.4 Libpcap Data							
Preamble Sink Detected Bursts	Total Packets Transmitted		100.00%	1912	Packet Lengths			Packet Count	Average Val (B)	Min Val (B)	Max Val (B)
	Total Packets Detected		99.95%	1911 <th>Total Packets</th> <td>100.00%</td> <td>1911</td> <td>58.57</td> <td>10</td> <td>90</td>		Total Packets	100.00%	1911	58.57	10	90
	Short Packets		0.00%	0		0-19	3.30%	63	12.25	10	18
	Bad CRC Packets		0.00%	0		20-39	2.51%	48	26.67	21	28
	Good CRC Packets		99.95%	1911		40-79	69.07%	1320	51.87	45	73
	Preambles Accepted		99.79%	1908		80-159	25.12%	480	86.27	80	90
	Preambles Rejected		0.16%	3							
	Total Processed Fingerprints		99.79%	1908		% Preamble's Fingerprinted		100.00%			

Table 5. Operational Performance Tables for Devices 5 to 8

Device: 5											
Time Span (sec)			25.262								
			Detected IEEE 802.15.4 Libpcap Data								
Preamble Sink Detected Bursts	Total Packets Transmitted		100.00%	1912	Packet Lengths			Packet Count	Average Val (B)	Min Val (B)	Max Val (B)
	Total Packets Detected		100.00%	1912		Total Packets	100.00%	1912	58.55	10	90
	Short Packets		0.00%	0		0-19	3.35%	64	12.25	10	18
	Bad CRC Packets		0.00%	0		20-39	2.51%	48	26.67	21	28
	Good CRC Packets		100.00%	1912		40-79	69.04%	1320	51.87	45	73
	Preambles Accepted		100.00%	1912		80-159	25.10%	480	86.27	80	90
	Preambles Rejected		0.00%	0							
Total Processed Fingerprints			100.00%	1912 <th colspan="2">% Preamble's Fingerprinted</th> <td colspan="4">100.00%</td>	% Preamble's Fingerprinted		100.00%				

Device: 6											
Time Span (sec)			25.201								
			Detected IEEE 802.15.4 Libpcap Data								
Preamble Sink Detected Bursts	Total Packets Transmitted		100.00%	1912	Packet Lengths			Packet Count	Average Val (B)	Min Val (B)	Max Val (B)
	Total Packets Detected		99.95%	1911		Total Packets	100.00%	1911	58.57	10	90
	Short Packets		0.00%	0		0-19	3.30%	63	12.25	10	18
	Bad CRC Packets		0.00%	0		20-39	2.51%	48	26.67	21	28
	Good CRC Packets		99.95%	1911		40-79	69.07%	1320	51.87	45	73
	Preambles Accepted		99.84%	1909		80-159	25.12%	480	86.27	80	90
	Preambles Rejected		0.10%	2							
	Total Processed Fingerprints		99.84%	1909		% Preamble's Fingerprinted		100.00%			

Device: 7											
Time Span (sec)			25.186								
			Detected IEEE 802.15.4 Libpcap Data								
Preamble Sink Detected Bursts	Total Packets Transmitted		100.00%	1912	Packet Lengths			Packet Count	Average Val (B)	Min Val (B)	Max Val (B)
	Total Packets Detected		100.00%	1912		Total Packets	100.00%	1912	58.55	10	90
	Short Packets		0.00%	0		0-19	3.35%	64	12.25	10	18
	Bad CRC Packets		0.00%	0		20-39	2.51%	48	26.67	21	28
	Good CRC Packets		100.00%	1912		40-79	69.04%	1320	51.87	45	73
	Preambles Accepted		99.95%	1911		80-159	25.10%	480	86.27	80	90
	Preambles Rejected		0.05%	1							
	Total Processed Fingerprints		99.90%	1910 <th colspan="2">% Preamble's Fingerprinted</th> <td colspan="5">99.95%</td>		% Preamble's Fingerprinted		99.95%			

Device: 8											
Time Span (sec)			25.112								
			Detected IEEE 802.15.4 Libpcap Data								
Preamble Sink Detected Bursts	Total Packets Transmitted		100.00%	1912	Packet Lengths			Packet Count	Average Val (B)	Min Val (B)	Max Val (B)
	Total Packets Detected		100.05%	1913		Total Packets	100.00%	1913	58.52	1	90
	Short Packets		0.05%	1		0-19	3.40%	65	12.08	1	18
	Bad CRC Packets		0.00%	0		20-39	2.51%	48	26.67	21	28
	Good CRC Packets		100.00%	1912		40-79	69.00%	1320	51.87	45	73
	Preambles Accepted		100.00%	1912		80-159	25.09%	480	86.27	80	90
	Preambles Rejected		0.05%	1							
	Total Processed Fingerprints		100.00%	1912		% Preamble's Fingerprinted	100.00%				

Table 6. Operational Performance Tables for Devices 9 and 10

Time Span (sec)				Device: 9							
25.182				Detected IEEE 802.15.4 Libpcap Data							
Preamble Sink Detected Bursts	Total Packets Transmitted		100.00%	1912	Packet Lengths			Packet Count	Average Val (B)	Min Val (B)	Max Val (B)
	Total Packets Detected		100.00%	1912		Total Packets	100.00%	1912	58.53	10	90
	Short Packets		0.00%	0		0-19	3.40%	65	12.32	10	18
	Bad CRC Packets		0.05%	1		20-39	2.51%	48	26.67	21	28
	Good CRC Packets		99.95%	1911		40-79	68.99%	1319	51.88	45	73
	Preambles Accepted		99.95%	1911		80-159	25.10%	480	86.27	80	90
	Preambles Rejected		0.05%	1							
	Total Processed Fingerprints		99.84%	1909		% Preamble's Fingerprinted		99.90%			

Device: 10											
Time Span (sec)				25.217							
25.217				Detected IEEE 802.15.4 Libpcap Data							
Preamble Sink Detected Bursts	Total Packets Transmitted		100.00%	1912	Packet Lengths			Packet Count	Average Val (B)	Min Val (B)	Max Val (B)
	Total Packets Detected		99.95%	1911		Total Packets	100.00%	1911	58.53	1	90
	Short Packets		0.05%	1		0-19	3.40%	65	12.08	1	18
	Bad CRC Packets		0.00%	0		20-39	2.51%	48	26.67	21	28
	Good CRC Packets		99.90%	1910		40-79	68.97%	1318	51.88	45	73
	Preambles Accepted		99.90%	1910		80-159	25.12%	480	86.27	80	90
	Preambles Rejected		0.05%	1							
	Total Processed Fingerprints		99.74%	1907		% Preamble's Fingerprinted		99.84%			

Table 7. Operational NRT Device Discrimination Joint PMF CM for $N_d=10$ Devices

RF-DNA PMF	Classified Device											Input Device Totals
	Device	1	2	3	4	5	6	7	8	9	10	
Input Device	1	8.24%	0.00%	0.00%	1.75%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	10.00%
	2	0.06%	9.69%	0.00%	0.00%	0.04%	0.00%	0.00%	0.00%	0.00%	0.21%	10.00%
	3	0.00%	0.00%	10.00%	0.00%	0.00%	0.00%	0.00%	0.01%	0.00%	0.00%	10.01%
	4	0.00%	0.00%	0.00%	9.99%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	9.99%
	5	0.51%	0.01%	0.00%	0.00%	9.50%	0.00%	0.00%	0.00%	0.00%	0.00%	10.01%
	6	0.00%	0.00%	0.00%	0.00%	0.00%	9.84%	0.01%	0.00%	0.05%	0.10%	10.00%
	7	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	9.87%	0.00%	0.13%	0.00%	10.00%
	8	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	10.01%	0.00%	0.00%	10.01%
	9	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.05%	0.01%	9.94%	0.00%	10.00%
	10	0.00%	0.00%	0.00%	0.00%	0.00%	0.14%	0.00%	0.00%	0.01%	9.84%	9.99%
Classified Device Totals		8.81%	9.69%	10.00%	11.75%	9.54%	9.99%	9.92%	10.02%	10.13%	10.15%	100.00%

Appendix C. Statistical File Examples

The three files below contain an example of the statistical data collected on Device 10 during the operational performance evaluation. The file's were limited to four-bursts due to the amount of data contained in each text file.

C.1 preamble_sink Statistical File Example

```
1
2
3 Collection started on 2019-01-12_17:58:11
4 Buffer size set to 4
5 Sample Rate at 4e+06 Samp/sec
6 Preamble sample length at 4e+06 Samp/sec is 640
7 Packet sample length at 4e+06 Samp/sec is 17024
8
9
10
11 Current Burst Sample Length: 2044
12 Buffer + Detected Burst Sample length: 12264
13 Total Packets Detected: 1
14 Short Packets: 0
15 Bad CRC Packets: 0
16 Good CRC Packets: 1
17 Total Preambles Processed
18 of 1 Total Packets Detected: 1
19 Preambles Accepted: 1
20 Preambles Rejected: 0
21
22
23
24 Current Burst Sample Length: 2044
25 Buffer + Detected Burst Sample length: 12264
26 Total Packets Detected: 2
27 Short Packets: 0
28 Bad CRC Packets: 0
29 Good CRC Packets: 2
30 Total Preambles Processed
31 of 2 Total Packets Detected: 2
32 Preambles Accepted: 2
33 Preambles Rejected: 0
34
35 ...
36 ...
37 ...
38 ...
39
40 Current Burst Sample Length: 2044
41 Buffer + Detected Burst Sample length: 12264
```

```

42  Total Packets Detected: 1910
43  Short Packets: 1
44  Bad CRC Packets: 0
45  Good CRC Packets: 1909
46  Total Preambles Processed
47  of 1910 Total Packets Detected: 1910
48  Preambles Accepted: 1909
49  Preambles Rejected: 1
50  =====
51
52  =====
53  Current Burst Sample Length: 2044
54  Buffer + Detected Burst Sample length: 12264
55  Total Packets Detected: 1911
56  Short Packets: 1
57  Bad CRC Packets: 0
58  Good CRC Packets: 1910
59  Total Preambles Processed
60  of 1911 Total Packets Detected: 1911
61  Preambles Accepted: 1910
62  Preambles Rejected: 1
63  =====

```

C.2 Dna_detector_ccf Device Statistical File Example

```

1
2  =====
3  Collection started on 2019-01-12_17:58:11
4  Sample Rate at 4e+06
5  Preamble sample length at 4e+06 Samp/sec is 640
6  Packet sample length at 4e+06 Samp/sec is 17024
7  =====
8
9  =====
10 Burst: 1
11 Device Most Likely: 10
12 Device Result Value: 1.193546
13 Total Valid Fingerprints: 1
14 Total Invalid Fingerprints: 0
15 Emitter_1 Count:0  Emitter_2 Count:0  Emitter_3 Count:0  Emitter_4 Count:0  Emitter_5 Count:0  Emitter_6 Count:0
    Emitter_7 Count:0  Emitter_8 Count:0  Emitter_9 Count:0  Emitter_10 Count:1
16  =====
17
18  =====
19 Burst: 2
20 Device Most Likely: 10
21 Device Result Value: 0.836388
22 Total Valid Fingerprints: 2
23 Total Invalid Fingerprints: 0
24 Emitter_1 Count:0  Emitter_2 Count:0  Emitter_3 Count:0  Emitter_4 Count:0  Emitter_5 Count:0  Emitter_6 Count:0
    Emitter_7 Count:0  Emitter_8 Count:0  Emitter_9 Count:0  Emitter_10 Count:2

```

```

25 -----
26 ...
27 ...
28 ...
29 ...
30 -----
31 Burst: 1906
32 Device Most Likely: 10
33 Device Result Value: 1.017697
34 Total Valid Fingerprints: 1906
35 Total Invalid Fingerprints: 0
36 Emitter_1 Count:0   Emitter_2 Count:0   Emitter_3 Count:0   Emitter_4 Count:0   Emitter_5 Count:0   Emitter_6 Count:27
    Emitter_7 Count:0   Emitter_8 Count:0   Emitter_9 Count:1   Emitter_10 Count:1878
37 -----
38
39 -----
40 Burst: 1907
41 Device Most Likely: 10
42 Device Result Value: 0.799719
43 Total Valid Fingerprints: 1907
44 Total Invalid Fingerprints: 0
45 Emitter_1 Count:0   Emitter_2 Count:0   Emitter_3 Count:0   Emitter_4 Count:0   Emitter_5 Count:0   Emitter_6 Count:27
    Emitter_7 Count:0   Emitter_8 Count:0   Emitter_9 Count:1   Emitter_10 Count:1879
46 -----
-----

```

C.3 Dna_detector_ccf Fingerprint Statistical File Example

```

1 -----
2 =====
3 Collection started on 2019-01-12_17:58:11
4 Sample Rate at 4e+06
5
6 Preamble sample length at 4e+06 Samp/sec is 640
7 Packet sample length at 4e+06 Samp/sec is 17024
8 Loaded Classification Matrix Dimensions: 90 by 9
9   -0.0179987   -0.0138279   0.0492875   -0.00519748   0.281197   0.166237   -0.0371916   0.108293   0.00584679
10   0.0519962   -0.0650138   0.0100148   0.092154   -0.133937   0.253971   0.0965857   0.206419   0.220455
11   0.0650591   -0.104349   -0.0112158   0.106646   0.178953   0.191529   0.169982   0.223511   0.123336
12   0.315404   -0.804263   0.557634   0.0858229   0.375554   -0.137245   0.165134   -0.192862   -0.0521735
13   0.0539755   -0.110696   0.0015731   -0.502993   0.413656   0.636204   -0.773287   -0.0236586   0.607816
14   0.205591   0.224312   0.601665   -0.664338   0.327243   0.352247   0.184607   -0.00797385   0.243865
15   -0.0288187   0.162438   -0.263308   0.128726   0.157559   -0.0410198   0.0192086   -0.0959234   0.201986
16   -0.00331162   0.00513426   0.0195337   -0.00185531   0.00113047   -0.00632859   0.0136545   0.00765422   -0.00407964
17   0.0122842   -0.0694236   0.151027   -0.0556066   -0.0619126   0.0200212   0.0101293   0.050921   -0.0970899
18   -0.00116661   0.00804955   0.00223123   0.00823627   -0.0051349   -0.010797   0.0226886   -0.007041   -0.000770946
19   0.00141869   0.00136686   0.00194486   0.00390335   0.00644027   0.0049   0.00280831   -0.0119721   -0.00876289
20   0.000303979   -0.00228553   -0.012385   0.00611358   -0.00362197   0.00808109   0.00849944   0.00209835   0.00443739
21   0.286092   0.142002   -0.0846356   -0.0348339   -0.0314808   -0.110885   -0.00964886   -0.00242085   -0.0020572
22   0.0601666   -0.0552247   0.0268359   0.0590591   -0.02967   -0.106003   -0.000755145   -0.0580027   0.180717
23   0.222119   -0.0793704   -0.0165477   0.105181   -0.214649   -0.0423035   -0.111313   0.0681685   0.0817924
24   -0.0640838   0.112785   0.0457627   0.0215456   0.0424662   -0.00647726   -0.00585118   -0.00699142   0.00176661

```

25	0.00662764	-0.0130145	0.00913062	-0.00687832	-0.00758337	-0.00270296	-0.00389443	0.0169814	-0.0098258
26	0.0467916	-0.0791779	-0.010964	-0.0188642	-0.0355097	-0.00243888	0.0025443	0.0285843	-0.020987
27	-0.0139836	0.00224475	-0.000782094	0.0106473	0.0162947	-0.00468398	0.0170125	-0.000588703	-0.00173349
28	0.00135554	-0.00194579	0.00212064	-0.0018191	0.0174199	0.00213627	0.00131997	0.00053738	-0.00648701
29	0.006105	-0.00286979	0.00289305	0.00144902	0.00308971	-0.00172469	0.00372499	0.017194	-0.0023469
30	0.26679	0.13497	-0.0396221	-0.057125	0.0106326	-0.0174177	-0.0544295	0.189978	-0.10586
31	0.00702072	-0.0815799	0.0622956	0.0856106	-0.0553958	-0.160043	0.0644781	0.314954	-0.0852966
32	0.150562	-0.087497	0.0628471	0.110113	-0.0482736	-0.117914	-0.076226	0.204006	0.0174335
33	-0.0308837	0.0869382	0.0328359	0.0330014	0.0902179	-0.00517578	-0.0378798	-0.0905071	0.0464061
34	0.00485134	-0.00929916	0.00463606	-0.000319247	-0.000540768	0.00334702	-0.00240521	0.00700152	-0.00905908
35	0.0273823	-0.0562473	-0.0563315	-0.00707477	-0.0396406	0.0127766	0.0301294	0.0619016	0.00743245
36	0.00246855	0.0159585	0.00940016	0.00335823	-0.0222815	0.0153068	0.025499	-0.00229938	0.0106375
37	-0.00239028	-0.00152254	-0.0117177	0.00200379	0.00888086	0.00303322	0.00102708	-0.0122456	-0.0141718
38	0.000811983	0.00226911	0.0094978	-0.000587597	0.0064853	0.00228381	0.00781912	-0.011869	-0.0047277
39	0.270512	0.135969	-0.0894139	-0.0291822	0.0363417	-0.0140701	-0.097259	-0.000771328	0.126223
40	0.0700662	-0.0159998	0.0571228	0.0690439	0.070612	-0.136603	0.067192	0.0170464	0.121596
41	0.192584	-0.0612598	0.07531	0.0810668	-0.0239006	-0.120201	-0.0909007	0.0886095	0.0738793
42	-0.0440164	0.071479	0.0846799	-0.00196105	0.00153133	-0.0129619	-0.0358571	-0.0187779	-0.0511077
43	0.00015605	-0.00464072	0.00819676	-0.00194985	-0.00267484	-0.00492951	0.00089783	0.00487488	0.00335984
44	0.0371283	-0.0568256	-0.069413	-0.000454897	-0.013599	0.00491682	0.0295402	0.0305295	0.0380307
45	-0.0105563	0.0060682	0.0123607	-0.0047119	0.0322205	-0.00164861	0.00630031	0.0073825	0.053492
46	-0.00193289	-0.00316073	-0.010652	0.005202	0.00458249	-0.00503012	-0.00577202	0.0168298	-0.0334098
47	1.48892e-05	-0.00669837	-0.00543729	-0.0042167	0.00630061	-0.00349612	-0.0103318	0.00553268	-0.000708032
48	0.240667	0.140633	-0.0231852	-0.0583104	-0.00973171	-0.0220327	-0.0138669	0.0804568	0.0761459
49	0.0239535	-0.0094095	0.0184948	0.0529527	0.141669	-0.162518	0.00136283	0.213032	0.191085
50	0.158215	-0.0575029	-0.111884	0.0716962	0.0967758	-0.173811	-0.101035	0.102195	0.182563
51	-0.0360789	0.077554	0.0788549	0.0111567	0.0915742	0.00938974	-0.0511118	-0.0515102	-0.0885251
52	-0.0017018	0.0015205	-0.0125357	0.00628536	0.00468503	-0.00530977	0.00116916	0.00378986	0.00727305
53	0.027629	-0.0594598	-0.0626026	-0.0065738	-0.0565151	-0.00730848	0.0359538	0.0486606	0.0644957
54	-0.000883298	0.0168459	0.0213659	-0.00601562	-0.00753495	0.0097391	-0.00421009	0.0103229	0.00841235
55	0.00551755	-0.00206573	-0.010558	0.00683553	0.0119927	0.0116338	0.0252045	-0.00351846	0.0224415
56	-0.000548334	-0.00396358	-0.0169977	0.00132838	0.0194445	-0.00063864	-0.0159161	-0.0106031	-0.0231037
57	0.220396	0.091192	-0.0901883	-0.0576103	-0.0433209	0.015877	-0.0996187	-0.0176815	0.194714
58	0.0326252	-0.0232445	0.0394201	0.0357308	0.0381709	-0.188134	-0.0077281	-0.0231976	0.0818933
59	0.195014	-0.0277167	-0.0183819	0.0916202	-0.0363662	-0.0526228	-0.205489	0.117058	-0.128334
60	-0.0304532	0.0706435	0.0785722	0.0159118	0.00751659	-0.0258178	-0.0277924	-0.0397856	0.0293018
61	0.000664844	0.000426695	-0.0120476	0.00667781	0.00561718	-0.00373585	0.00695089	0.00737165	0.0101539
62	0.0230686	-0.0432014	-0.0807095	-0.00363105	-0.00217805	0.0186467	0.0242959	0.0482847	0.0167243
63	-0.00518571	0.0104006	0.00164055	0.00504683	0.0235527	0.00532678	-0.0182492	-0.00432308	-0.0813744
64	0.00678387	-0.0101755	-0.000806586	0.00106888	0.00129832	0.012935	0.0327275	-0.0132255	0.0507408
65	0.00352698	-0.00639224	-0.00184754	-0.00200365	-0.0128195	0.011643	-0.00362152	-0.0224655	-0.0207719
66	0.263343	0.141425	-0.0185347	0.00940641	-0.00292029	0.023133	0.0143007	0.0942048	-0.00535658
67	-0.00394384	0.00353918	0.0248182	0.0055025	0.108469	-0.212352	-0.0504097	0.244767	-0.132579
68	0.154494	-0.017042	-0.0236012	0.0803724	0.0651184	-0.154514	-0.154937	0.0948392	-0.0341468
69	-0.0305	0.0436751	0.0646544	0.00291418	0.0690874	0.0260546	-0.0436364	-0.0356784	-0.0720672
70	0.000889159	0.000546561	-0.019191	0.00373607	-0.00234838	-0.0106462	-0.00191432	-0.00159631	0.0190787
71	0.0268579	-0.0344389	-0.0665503	0.00481281	-0.0558736	-0.0153351	0.0394623	0.0122939	0.0605611
72	-0.00273179	0.00969519	-3.53648e-05	0.00289651	0.0169789	-0.00535893	-0.0264398	-0.0343602	-0.0579403
73	0.0021705	0.00530617	-0.00371545	0.0107177	-0.0161151	0.0103277	0.0611815	0.0241568	0.013393
74	0.000575686	-0.00387589	-0.000264265	0.00456252	0.00632049	0.000761766	0.0102303	-0.00407106	-0.0116216
75	0.264313	0.0901443	-0.0999181	0.0726945	-0.125024	0.0621761	0.0548034	0.0817263	-0.0235207
76	-0.0222946	0.00601534	0.176647	0.196407	-0.0600623	-0.05713	0.315419	0.283528	-0.0445553
77	0.152544	0.0400634	0.184454	0.204482	-0.0491684	0.00177651	0.120559	0.174118	0.223844
78	-0.0236855	0.0607816	0.0519161	-0.00763007	-0.00999048	-0.0530825	-0.0590816	0.109305	-0.0587402

```
79 0.00163791 -0.00182925 0.00773689 -0.00100184 0.00115462 -0.00284512 0.00359856 0.00317279 0.000331348
80 0.010655 -0.0416392 -0.037304 0.00823299 0.014264 0.034567 0.0447979 -0.0692184 0.0569477
81 0.0110828 0.000542419 -0.0251615 0.0172514 -0.0223341 0.0279769 0.034671 -0.0281312 -0.0349511
82 -0.00451187 0.000598738 -0.00574578 -0.0034096 -0.0060025 0.000944656 -0.0376179 -0.0286255 -0.0613489
83 -0.00418865 0.00350385 0.0130416 -0.00248068 -0.00689904 -0.0054856 0.00642524 -0.00279389 0.0285479
84 0.351167 0.133245 0.0211305 0.292858 -0.470838 0.170856 0.107673 -0.554021 -0.267231
85 -0.0277754 0.00332057 -0.0596565 -0.0333592 0.122579 -0.0798377 -0.0312913 0.0319849 0.0233509
86 -0.0489184 -0.0156965 -0.0383189 -0.0130023 0.0934613 -0.0143486 0.019117 -0.109788 0.021779
87 0.0134799 0.0830591 0.139502 0.0166282 -0.00263858 0.0576215 -0.022693 -0.0928801 0.02145
88 -0.00119037 -0.00438005 -0.0119757 -0.00144486 0.00168315 0.00160842 0.00400101 0.00191221 0.00738228
89 -0.0111678 -0.0619892 -0.0966585 -0.0105575 0.00696647 -0.0310652 0.0264452 0.050225 -0.023025
90 0.00979592 0.00639145 -0.00898956 0.0124372 -0.015967 0.0120717 -0.0142736 0.0251983 -0.0872037
91 -0.0106373 0.0208767 0.0265617 0.00340747 0.00472755 0.00884205 -0.0415524 -0.0257788 0.0317484
92 -0.00248773 -0.000859742 0.00147527 0.000971058 -0.0295901 -0.00519992 0.0059612 0.0141136 -0.00987076
93 -0.0102819 -0.00200367 0.0133442 -0.00706274 0.015592 -0.00743293 -0.00384492 0.0181613 0.0181075
94 -0.00973775 -0.00370472 0.0112811 -0.00745067 0.0120673 -0.00657913 -0.00270948 0.0114451 0.0290378
95 0.00307149 0.00607471 0.00343391 0.00515173 0.00215804 -0.00903676 -0.00271573 0.00654477 0.0228145
96 -0.000742659 0.0428133 0.059984 0.00770555 0.0519115 -0.0231534 0.00277199 0.00406197 0.0590788
97 0.00513118 0.00392664 0.00314689 0.00428373 -0.00747823 -0.0040994 0.00397393 -0.0162887 -0.0158522
98 0.00383221 -0.0328871 -0.0266809 -0.00737345 -0.0574029 0.0313202 0.00279902 -0.0172337 -0.058471
99
100
101 Loaded Means Matrix Dimensions: 10 by 9
102 0.735917 -0.220484 0.397417 -0.169663 0.0318169 -0.103576 -0.126829 0.164935 -0.0316259
103 0.0508788 -0.966638 0.854638 0.216482 -0.241939 0.0732671 0.025962 -0.0406058 0.0204084
104 5.06297 0.238439 -0.407196 0.123757 -0.113997 0.0384649 0.100577 0.0214466 -0.0375498
105 1.15775 0.160057 0.0491624 -0.125593 0.19858 0.0499035 -0.281216 -0.122833 -0.069233
106 0.101127 -0.866081 -0.104861 -0.189655 -0.0845013 -0.278503 0.150856 -0.076027 0.0622165
107 -2.14498 -0.998712 -0.248582 -0.129318 0.0220415 0.199162 0.16485 0.0138251 -0.136249
108 -2.25908 0.990961 -0.376559 0.0520503 -0.56899 -0.0317732 -0.129809 0.00742225 -0.00259504
109 -0.487356 2.23551 0.365867 -0.0948389 0.161246 0.0659613 0.174637 -0.0125971 0.0448772
110 -1.78553 0.114444 -0.180992 0.308321 0.411908 -0.169985 -0.00329653 0.0136882 -0.0551149
111 -0.431695 -0.687494 -0.348894 0.00845715 0.183836 0.157079 -0.0757309 0.0307465 0.204865
112
113
114 Loaded Normalization Vector Dimensions: 1 by 90
115 13704.5 12.9373 2.04432 0.516744 8.52435 5.36929 7.41251e-13 5.17069 2.05593 307300
116 4.18024 2.29354 0.513156 12.445 8.2912 1.11898e-12 4.26269 1.41332 300307 4.28693
117 2.29232 0.513107 12.647 8.54486 1.13372e-12 4.37296 1.4717 296657 4.48851
118 2.33211 0.510407 12.5228 8.4741 1.15082e-12 4.36407 1.48042 294089 4.26038 2.32718
119 0.511454 12.4578 8.44662 1.14888e-12 4.33983 1.46938 285441 4.18313 2.32924
120 0.512439 12.5217 8.3969 1.15273e-12 4.34458 1.46239 293789 4.23062 2.30567 0.51427
121 12.6243 8.50782 1.1198e-12 4.30189 1.45054 295718 4.27182 2.61244 0.626118
122 11.7778 6.28912 1.1261e-12 4.02128 1.43511 303840 4.86541 3.02887 0.401845 41.5998
123 19.5941 1.07547e-12 3.98964 1.34588 356727 3.20272 2.59439 0.6481 4.3408
124 5.69092 1.15874e-12 3.69243 1.54401
125
126
127
128 Loaded X-Offset Vector Dimensions: 1 by 90
129 0.00054359 -2.63705 6.29743 24.0083 -0.691296 -0.662844 1.17801e+13 -0.0467497 -0.940394 1.53269e-05
130 -0.286624 -0.219566 31.0208 -0.482996 -0.928594 9.90511e+12 -0.293949 -1.2065 1.56513e-05 -0.34421
131 -0.186094 31.0305 -0.479196 -0.934573 9.92854e+12 -0.292703 -1.22004 1.5736e-05 -0.331072 -0.199421
132 31.0431 -0.478048 -0.936341 9.88561e+12 -0.288938 -1.21683 1.60963e-05 -0.302808 -0.231789 31.0776
133 -0.481237 -0.935591 9.9424e+12 -0.286381 -1.21859 1.58991e-05 -0.286129 -0.254877 31.0303 -0.48204
```

```

-0.932435 9.83179e+12 -0.292364 -1.21741 1.58548e-05 -0.280131 -0.241095 31.0449 -0.479703 -0.932668
9.93471e+12 -0.289403 -1.21405 1.63669e-05 -0.232444 -0.347459 23.977 -0.661172 -0.771309 9.74863e+12
-0.211929 -1.21972 1.83216e-05 -0.209075 -0.514383 22.562 -0.581289 -0.685192 9.50765e+12 -0.364518
-1.06211 1.4744e-05 -0.0986884 -0.273749 7.71534 0.404481 -0.64956 9.93909e+12 0.1387 -1.28045
120
121
122
123
124 Burst: 1
125 Fingerprints: 0.000484142, -2.6124, 6.14313, 23.8671, -0.688413, -0.754609, 1.12116e+13, 0.0285509, -1.26072, 1.44675e-05,
-0.302506, -0.564842, 30.6739, -0.499393, -0.933092, 1.0682e+13, 0.183254, -0.253016, 1.65275e-05, -0.0912049, -0.400581,
30.1432, -0.487574, -0.950747, 9.46865e+12, -0.212311, -1.41905, 1.1274e-05, -0.759928, 0.324143, 30.4293, -0.48934,
-0.96954, 8.71915e+12, -0.294737, -1.57909, 1.48742e-05, -0.169879, -0.907481, 30.1696, -0.494958, -0.964221, 8.80993e+12,
-0.246201, -1.62185, 1.50447e-05, -0.181404, -0.869718, 30.6034, -0.468557, -0.992598, 9.91971e+12, -0.536651, -0.706624,
1.42401e-05, -0.502983, -0.262833, 30.081, -0.495264, -0.940768, 1.06809e+13, 0.271238, 0.164766, 1.35402e-05, -0.209981,
-0.404162, 23.6465, -0.653124, -0.801539, 9.10998e+12, -0.173066, -1.52532, 1.6674e-05, -0.100058, -0.892026, 21.9966,
-0.608192, -0.6692, 8.41629e+12, -0.34357, -1.43081, 1.26325e-05, 0.10747, -0.390506, 7.15869, 0.209104, -0.935953,
9.57973e+12, 0.00453088, -1.49315
126 DNA Results: 2.21695, 1.8652, 6.1014, 2.58573, 1.38208, 1.57145, 2.21141, 3.24969, 1.75002, 1.19355
127
128
129
130 Burst: 2
131 Fingerprints: 0.000547583, -2.52409, 5.56325, 24.4746, -0.680105, -0.775479, 1.19888e+13, 0.19656, -0.803937, 1.46222e-05,
-0.288606, -0.866538, 30.4022, -0.48049, -0.95131, 8.72298e+12, -0.297941, -1.55877, 1.51409e-05, -0.204184, -0.379061,
30.4292, -0.462755, -0.977408, 9.23294e+12, -0.240453, -1.62684, 1.32202e-05, -0.0847037, -0.391554, 30.3984, -0.475175,
-0.97645, 9.91199e+12, -0.521736, -0.749453, 1.58215e-05, -0.535532, -0.700479, 30.2888, -0.475259, -0.966004, 1.05999e
+13, 0.238615, -0.00744286, 1.5033e-05, -0.0767738, -0.521628, 30.4633, -0.481944, -0.951706, 9.01269e+12, -0.255107,
-1.46575, 1.30094e-05, -0.0498481, -0.817803, 30.5269, -0.464966, -0.954613, 8.77376e+12, -0.321888, -1.54646, 1.62344e
-05, -0.139222, -0.816018, 23.4088, -0.652444, -0.811476, 8.85937e+12, -0.147261, -1.62803, 1.5216e-05, -0.208534,
-0.495012, 21.8262, -0.564223, -0.700463, 9.97947e+12, -0.851366, 0.619849, 1.35183e-05, 0.396515, -0.502405, 7.0404,
0.197694, -0.984181, 9.40135e+12, -0.0638478, -1.67356
132 DNA Results: 2.08888, 1.68626, 6.11482, 2.54044, 1.17361, 1.38759, 2.40474, 3.38157, 1.63051, 0.836388
133
134 ...
135 ...
136 ...
137 ...
138
139 Burst: 1906
140 Fingerprints: 0.000462542, -2.6257, 6.3197, 24.1031, -0.602659, -0.874896, 1.01729e+13, 0.0660602, -1.25633, 1.30733e-05,
-0.590974, 0.0325582, 30.684, -0.442804, -1.02646, 1.03071e+13, 0.297578, 0.119069, 1.33052e-05, -0.249575, 0.355452,
30.9752, -0.411574, -1.06948, 9.37987e+12, -0.202414, -1.53203, 1.39997e-05, -0.544521, 0.492708, 31.0575, -0.421406,
-1.02555, 8.82965e+12, -0.29291, -1.64249, 1.4321e-05, -0.0658905, -0.489685, 31.0135, -0.436565, -1.03532, 9.03493e+12,
-0.252445, -1.72479, 1.52498e-05, -0.626062, 0.00840969, 30.9541, -0.43263, -1.04646, 9.73887e+12, -0.486588, -0.896012,
1.07034e-05, -0.635349, 0.137133, 30.662, -0.447852, -1.01515, 1.06774e+13, 0.236712, -0.0599057, 1.80834e-05, -0.568642,
-0.0407675, 23.887, -0.584813, -0.907803, 9.16933e+12, -0.125612, -1.67471, 1.66455e-05, -0.325289, -0.110003, 22.8192,
-0.572374, -0.719272, 9.36533e+12, -0.375178, -1.45231, 1.62984e-05, -0.206892, -0.742023, 8.07163, 0.647993, -0.544134,
8.99095e+12, 0.129211, -1.73623
141 DNA Results: 2.08485, 1.76511, 6.14833, 2.52914, 1.26536, 1.36921, 2.20284, 3.24735, 1.5724, 1.0177
142
143
144
145 Burst: 1907
```

146 Fingerprints: 0.00058104, -2.51017, 5.37284, 24.6607, -0.64132, -0.844598, 1.14472e+13, -0.0216109, -1.32158, 1.0891e-05,
-0.00529928, -0.56346, 30.943, -0.457444, -0.976167, 1.0606e+13, 0.123451, -0.569941, 1.21239e-05, -0.398005, 0.898912,
30.6674, -0.455931, -0.972589, 9.29122e+12, -0.283727, -1.76513, 1.1473e-05, -0.370064, -0.524713, 30.3333, -0.462905,
-0.987202, 8.58564e+12, -0.283802, -1.78161, 1.30263e-05, 0.0928206, -0.146007, 30.649, -0.466955, -0.983474, 9.19419e+12,
-0.254549, -1.7973, 1.1591e-05, -0.204375, -0.241568, 30.6361, -0.441498, -1.01157, 1.04161e+13, -0.562115, -0.6741,
1.39238e-05, 0.24214, -0.21074, 30.7725, -0.459178, -0.998894, 1.04556e+13, 0.138324, -0.600393, 1.16295e-05, 0.0705197,
-0.705736, 23.6096, -0.639583, -0.813336, 9.06226e+12, -0.14256, -1.8012, 1.65229e-05, 0.064139, -0.692495, 22.143,
-0.58209, -0.674799, 8.29928e+12, -0.295043, -1.75859, 1.48085e-05, -0.124188, -0.0900506, 6.1492, 0.545806, -0.355956,
9.12795e+12, 0.121215, -1.87803
147 DNA Results: 2.07239, 1.70087, 6.09538, 2.48023, 1.24685, 1.45615, 2.29295, 3.28073, 1.5466, 0.799719
148

Appendix D. Source Code

The air monitor's Python code was not included because it can easily be reproduced using *gnuradio-companion*. The *packet_sink* and *Dna_detector_ccf* blocks source code were created using *gr_modtool*. Some pure virtual public functions were added to the header files to give the user access to the private implementation files as setters and getters. The SWIG generated Python functions will have access to the virtual public functions. The *packet_sink* and *Dna_detector_ccf* XML block files were modified to provide the users the ability to save and load files with a GUI instead of just a name; a drop down menu was added for the debug option; and several "On"/"Off" and "Yes"/"No" drop downs for the options for the blocks.

The classes listed below were broken down to make them a little more readable. Even though everything compiled without any flags or errors, there were runtime issues with the SWIG generated *packet_sink* Python code because the DNA buffer class, base packet class, IEEE 802.15.4 packet class, file read implementation class, file store base class, and file store implementation classes were not being detected. Due to time constraints and my inexperience with SWIG, the class definitions were inserted into the *packet_sink_impl.h* header file while the source code was inserted into *packet_sink_impl.cc* to make everything work as intended.

D.1 Packet Sink Block Source Code

D.1.1 Packet Sink C++ Code.

```
1  /*****
   ***
2  *preamble_sink.cc class
3  *****/
   **/
4  /* -- C++ -- */
5  /*
6   * Copyright 2018 gr-ieee802_15_4 author.
7   *
```

```

8  * This is free software; you can redistribute it and/or modify
9  * it under the terms of the GNU General Public License as published by
10 * the Free Software Foundation; either version 3, or (at your option)
11 * any later version.
12 *
13 * This software is distributed in the hope that it will be useful,
14 * but WITHOUT ANY WARRANTY; without even the implied warranty of
15 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
16 * GNU General Public License for more details.
17 *
18 * You should have received a copy of the GNU General Public License
19 * along with this software; see the file COPYING. If not, write to
20 * the Free Software Foundation, Inc., 51 Franklin Street,
21 * Boston, MA 02110-1301, USA.
22 */
23
24 #ifndef INCLUDED_IEEE802_15_4_PREAMBLE_SINK_H
25 #define INCLUDED_IEEE802_15_4_PREAMBLE_SINK_H
26
27 #include <ieee802_15_4/api.h>
28 #include <gnuradio/block.h>
29
30 namespace gr {
31     namespace ieee802_15_4 {
32         /*!
33          * SAVE enable/disable saving file statistics
34          * or everything else.
35          */
36         enum SAVE{
37             NO = 0,
38             YES = 1
39         };
40
41         /*!
42          * CRC will be used to determine if either crc verified data will be output
43          * or everything else.
44          */
45         enum CRC{
46             ALL_DATA = 0,
47             CRC_ONLY = 1
48         };
49
50         /*!
51          * DebugType will be used to determine the level of debugging outputs that will be enabled/disabled
52          */
53         enum DebugType{
54             ALL_OFF = 0,
55             CHIP_ON = 1,
56             PACKET_ON = 2,
57             USRP_ON = 4,
58             CHIP_PACKET_ON = 3,
59             CHIP_USRP_ON = 5,
60             PACKET_USRP_ON = 6,
61             ALL_ON = 7

```

```

62     };
63
64     /*!
65     * Class: preamble_sink
66     * Author: Schmid, Thomas; Bloessl, Bastian; Cruz, Frankie
67     * Description: the class is directly derived from Bastian Bloessl's
68     *               packet_sink class from his IEEE802.15.4 It is the same
69     *               code with a few minor modifications that allow the RX
70     *               802.15.4 preamble to be sent once it has been detected
71     * Generated: Thurs Nov 8 15:30:18 2018
72     * Modified: Sat Feb 2 08:00:23 2018
73     * Build Version: 1.16
74     * Changes: -Fixed short byte statistic
75     *           -Cleaned code up a bit
76     *           -Fixed Buffer placement...
77     */
78     class IEEE802_15_4_API preamble_sink : virtual public gr::block
79     {
80     public:
81         typedef boost::shared_ptr<preamble_sink> sptr;
82
83         /*!
84         * Make preambe_sink
85         *
86         * param samp_rate is the sample rate of the system
87         * param buffer_size is an array length consisting of previous bursts
88         * param threshold is the error threshold of the detected chips
89         * param debug is the debug output message setting
90         * param crc is the crc verified vs non-verified acceptance setting
91         * param burst_trigger will set the starting threshold of detection for normalized preamble
92         * param save_wi enable/disable a file save of the packets
93         * param save_file is the loction of where the file will be saved
94         */
95         static sptr make(float samp_rate, unsigned int threshold, unsigned int buffer_size,
96                          float burst_trigger, DebugType debug, CRC crc,
97                          SAVE save_stats_switch, std::string save_stats_file);
98
99         virtual void set_threshold(const unsigned int &threshold) = 0;
100        virtual void set_samp_rate(const float &samp_rate) = 0;
101        virtual void set_buffer_size(const unsigned int &buffer_size) = 0;
102        virtual void set_burst_trigger(const float &burst_trigger) = 0;
103        virtual void set_debug(const DebugType &debug) = 0;
104        virtual void set_crc(const CRC &crc) = 0;
105        virtual void set_save_stats_switch(const SAVE &save_stats_switch) = 0;
106        virtual void set_save_stats_file(const std::string &save_stats_file) = 0;
107
108        virtual unsigned int get_threshold() const = 0;
109        virtual float get_samp_rate() const = 0;
110        virtual unsigned int get_buffer_size() const = 0;
111        virtual float get_burst_trigger() const = 0;
112        virtual DebugType get_debug() const = 0;
113        virtual SAVE get_save_stats_switch() const = 0;
114        virtual std::string get_save_stats_file() const = 0;
115    };

```

```

116
117     } // namespace ieee802_15_4
118 } // namespace gr
119 #endif /* INCLUDED_IEEE802_15_4_PREAMBLE_SINK_H */
120
121 /*****
122  *preamble_sink_impl.h implementation class definitions
123 *****/
124 /* -- c++ -- */
125 /*
126  * Copyright 2018 gr-ieee802_15_4 author.
127  *
128  * This is free software; you can redistribute it and/or modify
129  * it under the terms of the GNU General Public License as published by
130  * the Free Software Foundation; either version 3, or (at your option)
131  * any later version.
132  *
133  * This software is distributed in the hope that it will be useful,
134  * but WITHOUT ANY WARRANTY; without even the implied warranty of
135  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
136  * GNU General Public License for more details.
137  *
138  * You should have received a copy of the GNU General Public License
139  * along with this software; see the file COPYING. If not, write to
140  * the Free Software Foundation, Inc., 51 Franklin Street,
141  * Boston, MA 02110-1301, USA.
142  */
143
144 #ifndef INCLUDED_IEEE802_15_4_PREAMBLE_SINK_IMPL_H
145 #define INCLUDED_IEEE802_15_4_PREAMBLE_SINK_IMPL_H
146
147 #include <ieee802_15_4/preamble_sink.h>
148 #include <cstdio>
149 #include <errno.h>
150 #include <sys/types.h>
151 #include <sys/stat.h>
152 #include <fcntl.h>
153 #include <stdexcept>
154 #include <cstring>
155 #include <gnuradio/blocks/count_bits.h>
156 #include <iostream>
157 #include <tuple>
158 #include <cmath>
159 #include <sstream>
160 #include <iostream>
161 #include <boost/thread.hpp>
162 #include <boost/thread/mutex.hpp>
163 #include <typeinfo>
164 #include <iomanip>
165 #include <ctime>
166
167 // win32 (mingw/msvc) specific

```

```

168 #ifdef HAVE_IO_H
169 #include <io.h>
170 #endif
171 #ifdef O_BINARY
172 #define OUR_O_BINARY O_BINARY
173 #else
174 #define OUR_O_BINARY 0
175 #endif
176
177 // should be handled via configure
178 #ifdef O_LARGEFILE
179 #define OUR_O_LARGEFILE O_LARGEFILE
180 #else
181 #define OUR_O_LARGEFILE 0
182 #endif
183
184 namespace gr {
185     namespace ieee802_15_4 {
186         /*!
187          * Class: preamble_sink_impl
188          * Author: Schmid, Thomas; Bloessl, Bastian; Cruz, Frankie
189          * Description: the class is the implementation file of
190          *                 preamble_sink.h. See preamble_sink.h
191          *                 for more information about this class
192          * Generated: Thurs Nov 8 15:30:18 2018 Wed Nov 28 10:30:11 2018
193          * Modified: Dec 01 13:30:01 2018
194          * Build Version: 1.6
195          * Changes: See preamble_sink.h
196          */
197
198         // this is the mapping between chips and symbols if we do
199         // a fm demodulation of the O-QPSK signal. Note that this
200         // is different than the O-QPSK chip sequence from the
201         // 802.15.4 standard since there there is a translation
202         // happening.
203         // See "CMOS RFIC Architectures for IEEE 802.15.4 Networks",
204         // John Notor, Anthony Caviglia, Gary Levy, for more details.
205         static const unsigned int CHIP_MAPPING[] = {
206             1618456172,
207             1309113062,
208             1826650030,
209             1724778362,
210             778887287,
211             2061946375,
212             2007919840,
213             125494990,
214             529027475,
215             838370585,
216             320833617,
217             422705285,
218             1368596360,
219             85537272,
220             139563807,
221             2021988657};

```

```

222
223
224 static const int MAX_PKT_LEN    = 128 - 1; // remove header and CRC
225 static const int MAX_LQI_SAMPLES = 8; // Number of chip correlation samples to take
226
227 class preamble_sink_impl : public preamble_sink
228 {
229     private:
230         enum {STATE_SYNC_SEARCH, STATE_HAVE_SYNC, STATE_HAVE_HEADER} d_state;
231
232         unsigned int    d_sync_vector;           // 802.15.4 standard is 4x 0 bytes and 1x0xA7
233         unsigned int    d_threshold = 10;        // how many bits may be wrong in sync vector
234
235         unsigned int    d_shift_reg;             // used to look for sync_vector
236         int             d_preamble_cnt;          // count on where we are in preamble
237         int             d_chip_cnt;              // counts the chips collected
238
239         unsigned int    d_header;                // header bits
240         int             d_headerbitlen_cnt;      // how many so far
241
242         unsigned char    d_packet[MAX_PKT_LEN];  // assembled payload
243         unsigned char    d_packet_byte;          // byte being assembled
244         int             d_packet_byte_index;     // which bit of d_packet_byte we're working on
245         int             d_packetlen;             // length of packet
246         int             d_packetlen_cnt;         // how many so far
247         int             d_payload_cnt;           // how many bytes in payload
248
249         unsigned int    d_lqi;                   // Link Quality Information
250         unsigned int    d_lqi_sample_count;
251
252         float d_burst_trigger = 0.3f;            // default preamble burst trigger
253         float d_samp_rate = 4e6f;                // block system's sample rate
254         int d_chip_debug = 0;                     // very verbose output for almost each sample
255         int d_pack_debug = 0;                     // less verbose output for higher level debugging
256         int d_usrp_debug = 0;                     // USRP verbose output for debugging
257         int d_crc_set = 0;                        // CRC data input setting
258         int d_save_switch_set = 0;                // Save switch input setting
259         int d_crc_good_payload = 0;               // good data packet counter
260         int d_crc_bad_payload = 0;                // bad data packet counter
261         int d_short_payload = 0;                  // short payload counter
262         int d_recent_packet_length = 0;           // sanity check for packet length
263         int d_removed_preambles = 0;              // detected preambles that were too short to process
264         int d_sent_preambles = 0;                 // detected preambles that were sent for RFDNA processing
265         bool d_preamble_processed = false;        // Activates preamble processing
266         unsigned int d_buffer_size;                // previous storage buffer length
267         bool d_prev_buffer_activated = false;
268
269         DebugType d_debug = ALL_OFF;              // Default debug state
270         CRC d_crc = ALL_DATA;                     // All on by default
271         ieee802_15_4_packet d_phy_802_15_4;      // ieee802.15.4 packet with size info
272
273         SAVE d_save_stats_switch = NO;            // off by default
274         std::string d_save_stats_file;             // file location where to save stats
275         std::stringstream d_msg_stream;           // handles string messages

```

```

276     std::shared_ptr<file_store_impl> d_stats_writer; // for statistics output file
277
278     // FIX ME
279     char buf[256]; // PDDU Packet Buffer
280     dna_buffer d_prev_usrp_collect_buffer;           // previous run usrp buffered data
281     dna_buffer d_current_preamble_buffer;           // current run's detected buffer data for crc processing
282     dna_buffer d_crc_preamble_buffer;               // crc evaluated buffered data
283
284     // enter_search, enter_have_sync,
285     // and enter_have_header are state
286     // functions used to set the private
287     // variables based on if the RX
288     // signal is searching, found, and
289     // parsed ieee 802.15.4 packets
290     void enter_search();
291     void enter_have_sync();
292     void enter_have_header(const int &payload_len);
293
294 public:
295     preamble_sink_impl(float samp_rate, unsigned int threshold, unsigned int buffer,
296                        float burst_trigger, DebugType debug, CRC crc,
297                        SAVE save_stats_switch, std::string save_stats_file);
298     ~preamble_sink_impl();
299
300     // Where all the action really happens
301     void forecast (int noutput_items, gr_vector_int &ninput_items_required);
302
303     int general_work(int noutput_items,
304                     gr_vector_int &ninput_items,
305                     gr_vector_const_void_star &input_items,
306                     gr_vector_void_star &output_items);
307
308     // decode_chips takes in the value of the RX
309     // chips as an integer and performs a XOR comparison
310     // between the CHIP_MAPPING vector to return a 1-symbol
311     // via a byte equivalent to 1 of the 16 mapped constellation
312     // values.
313     unsigned char decode_chips(const unsigned int &chips);
314
315     // slice will be used to determine if an input bit is
316     // 1 or 0 for the shift register
317     int slice(const float &x);
318
319     // setter & getter for error threshold
320     void set_threshold(const unsigned int &threshold);
321     unsigned int get_threshold() const;
322
323     // getters and setters to get/set sample rate
324     void set_samp_rate(const float &samp_rate);
325     float get_samp_rate() const;
326
327     // getters and setters for buffer size
328     void set_buffer_size(const unsigned int &buffer);
329     unsigned int get_buffer_size() const;

```

```

330
331 //getters and setters for burst trigger
332 void set_burst_trigger(const float &burst_trigger);
333 float get_burst_trigger() const;
334
335 // getters and setters to get/set debug settings
336 void set_debug(const DebugType &debug);
337 DebugType get_debug() const;
338
339 // getters and setters to get/set crc
340 // verified packets only or everything
341 void set_crc(const CRC &crc);
342 CRC get_crc() const;
343
344 // getters and setters for stat_file functions
345 void set_save_stats_switch(const SAVE &save_stats_switch);
346 void set_save_stats_file(const std::string &save_stats_file);
347 SAVE get_save_stats_switch() const;
348 std::string get_save_stats_file() const;
349
350 // function used in mac.cc from gr-ieee802-15-4
351 // to verify data validity of raw data
352 uint16_t crc16(const char *buf, const int &len);
353
354 // shortcut to clear stringstream
355 void clear_stream();
356
357 // generic print print function that for vector
358 // inputs
359 void vector_print(const std::string &label, const std::vector<int> &input);
360
361 // returns a trigger-set/sample adjusted preamble from the passed in buffer
362 std::tuple<dna_buffer, bool> normalize_preamble(const dna_buffer &buffer,
363                                               const int &max_return_size,
364                                               const float &burst_trigger);
365
366 // NOTE: Didn't finish, but will if I have time...
367 // Filter_burst is going to filter the absolute magnitude of the burst with an n-taps.
368 // The magnitude pulse scaled and normalized to 1 and the index point where the filtered
369 // burst exceeds the user-given threshold
370 std::tuple<dna_buffer, int> filter_burst(const dna_buffer &buffer, const int &filter_taps);
371 };
372
373 } // namespace ieee802_15_4
374 } // namespace gr
375 #endif /* INCLUDED_IEEE802_15_4_PREAMBLE_SINK_IMPL_H */
376
377 /*****
378  *preamble_sink_impl.cc Source Code
379  *****/
380 #include <gnuradio/io_signature.h>
381 #include "preamble_sink_impl.h"

```

```

382 // #include <gnuradio/block.h>
383
384 namespace gr {
385     namespace ieee802_15_4 {
386         /*****
387         * preamble_sink source code section
388         *****/
389         preamble_sink::sptr
390         preamble_sink::make(float samp_rate, unsigned int threshold, unsigned int buffer_size,
391                             float burst_trigger, DebugType debug, CRC crc,
392                             SAVE save_stats_switch, std::string save_stats_file)
393         {
394             return gnuradio::get_initial_sptr
395                 (new preamble_sink_impl(samp_rate, threshold, buffer_size,
396                                         burst_trigger, debug, crc,
397                                         save_stats_switch, save_stats_file));
398         }
399
400         /*****
401         * preamble_sink_impl source code section
402         *****/
403         /*
404         * The private constructor
405         */
406         preamble_sink_impl::preamble_sink_impl(float samp_rate, unsigned int threshold, unsigned int buffer_size,
407                                                 float burst_trigger, DebugType debug, CRC crc,
408                                                 SAVE save_stats_switch, std::string save_stats_file)
409         : gr::block("preamble_sink",
410                     gr::io_signature::make(2, 2, sizeof(float), sizeof(gr_complex)),
411                     gr::io_signature::make(2, 2, sizeof(gr_complex), sizeof(gr_complex)))
412         {
413             d_sync_vector = 0xA7;
414             d_samp_rate = samp_rate;
415             d_threshold = threshold;
416             d_save_stats_file = save_stats_file;
417             d_burst_trigger = burst_trigger;
418             set_debug(debug);
419             set_crc(crc);
420             set_save_stats_switch(save_stats_switch);
421             set_buffer_size(buffer_size);
422
423             // Set sample rate for our packet size calculations
424             d_phy_802_15_4.set_sample_rate(d_samp_rate);
425
426             std::cout << "\nPreamble_sink sample rate set to " << samp_rate << std::endl;
427
428             // file name and save settings set here
429             if(d_save_stats_switch == SAVE::YES)
430             {
431                 // from stackoverflow.com, just a quick way to append time/date to a filename
432                 auto t = std::time(nullptr);
433                 auto tm = *std::localtime(&t);
434                 d_msg_stream << "_" << std::put_time(&tm, "%d%b%y_%R") << ".txt";
435                 d_save_stats_file.append(d_msg_stream.str());

```

```

436     clear_stream();
437
438     std::cout << "\nFile stats set to save at " << d_save_stats_file << ".\n" << std::endl;
439     d_stats_writer = std::make_shared<file_store_impl>(sizeof(char), d_save_stats_file.c_str(),
440                                                     false, false);
441
442     d_msg_stream << "\n===== " << "\n";
443     d_msg_stream << "Collection started on " << std::put_time(&tm, "%F_%T");
444     d_msg_stream << "\nBuffer size set to " << d_buffer_size;
445     d_msg_stream << "\nSample Rate at " << samp_rate << " Samp/sec";
446     d_msg_stream << "\nPreamble sample length at " << samp_rate << " Samp/sec is " << d_phy_802_15_4.
        get_preamble_len_samples();
447     d_msg_stream << "\nPacket sample length at " << samp_rate << " Samp/sec is " << d_phy_802_15_4.get_packet_len_samples()
        ;
448     d_msg_stream << "\n===== " << "\n";
449     d_stats_writer->store_data(d_msg_stream.str().size(), (void*)d_msg_stream.str().c_str());
450     clear_stream();
451 }
452
453 // Link Quality Information
454 d_lqi = 0;
455 d_lqi_sample_count = 0;
456
457 //need to convert to a int for this to work
458 set_output_multiple((int)round(d_phy_802_15_4.get_sym_to_chip_mult()));
459 set_min_output_buffer(1,40881); //hard coded for now to ensure a proper output for passthrough usrp data...need to fix
    later
460
461 if(d_chip_debug)
462 {
463     fprintf(stderr, "syncvec: %x, threshold: %d\n", d_sync_vector, d_threshold), fflush(stderr);
464 }
465 enter_search();
466
467 message_port_register_out(pmt::mp("out"));
468 }
469
470 /*
471  * Our virtual destructor.
472  */
473 preamble_sink_impl::~preamble_sink_impl()
474 {
475 }
476
477 void
478 preamble_sink_impl::forecast (int noutput_items, gr_vector_int &ninput_items_required)
479 {
480     /*****Special Note*****/
481     ***Need to see if this is adjustable before and after compilation for different Samp Rates***
482     *****/
483
484     // need to fix ieee802_15_4_packet class to better handle inputs
485     // original implementation was blowing up block
486     // now this needs to be hard coded for the moment

```

```

487     ninput_items_required[0] = 1022;
488     ninput_items_required[1] = 2044;
489 }
490
491 int
492 preamble_sink_impl::general_work (int noutput_items,
493                                   gr_vector_int &ninput_items,
494                                   gr_vector_const_void_star &input_items,
495                                   gr_vector_void_star &output_items)
496 {
497     const float *inbuf = (const float *) input_items[0]; // input stream M & M stream of floats for detection
498     const gr_complex *in_rx = (const gr_complex *) input_items[1]; // input stream of RX data
499
500     int ninput = ninput_items[0]; // input number of detected float data
501     int ninput_rx = ninput_items[1]; // input number of RX data
502
503     gr_complex *out_rx = (gr_complex *) output_items[0]; // output RX stream for detected preambles
504     gr_complex *out_usrp = (gr_complex *) output_items[1]; //input to output raw stream
505
506     int count = 0;
507     static std::vector<int> preamble_marker; // used to store preamble markers per each general_work iteration
508     static std::vector<int> sync_marker; // used to store the sync markers per general_work iteration
509     static std::vector<int> header_marker; // used to store the header markers per general_work iteration
510
511     int i = 0;
512
513     if (d_chip_debug)
514         fprintf(stderr, ">>> Entering state machine\n"), fflush(stderr);
515
516     // The burst buffer is going to determine how many previous bursts will be "searched"
517     // through once a proper packet has been detected. If the buffer is bigger than the
518     // user set limit, the FIFO structure will pop off the front burst samples
519     if (d_prev_usrp_collect_buffer.get_burst_length() > d_buffer_size)
520     {
521         d_prev_usrp_collect_buffer.pop_front_burst_samples();
522     }
523     while (count < ninput) {
524         switch (d_state) {
525
526             case STATE_SYNC_SEARCH: // Look for sync vector
527                 if (d_chip_debug)
528                     fprintf(stderr, "SYNC Search, ninput=%d syncvec=%x\n", ninput, d_sync_vector), fflush(stderr);
529
530                 while (count < ninput) {
531
532                     if (slice(inbuf[count++]))
533                         d_shift_reg = (d_shift_reg << 1) | 1;
534                     else
535                         d_shift_reg = d_shift_reg << 1;
536
537                     if (d_preamble_cnt > 0) {
538                         d_chip_cnt = d_chip_cnt + 1;
539                     }
540

```

```

541 // The first if block synchronizes to chip sequences.
542 if(d_preamble_cnt == 0){
543     unsigned int threshold;
544     threshold = gr::blocks::count_bits32((d_shift_reg & 0x7FFFFFFF) ^ (CHIP_MAPPING[0] & 0x7FFFFFFF));
545     if(threshold < d_threshold) {
546         // fprintf(stderr, "Threshold %d d_preamble_cnt: %d\n", threshold, d_preamble_cnt);
547         // if ((d_shift_reg&0x7FFFFFFF) == (CHIP_MAPPING[0]&0x7FFFFFFF)) {
548         if (d_pack_debug)
549             fprintf(stderr, "Found 0 in chip sequence\n"), fflush(stderr);
550         // we found a 0 in the chip sequence
551         d_preamble_cnt++;
552         // fprintf(stderr, "Threshold %d d_preamble_cnt: %d\n", threshold, d_preamble_cnt);
553     }
554 } else {
555     // we found the first 0, thus we only have to do the calculation every 32 chips
556     if(d_chip_cnt == 32){
557         d_chip_cnt = 0;
558
559         if(d_packet_byte == 0) {
560             if (gr::blocks::count_bits32((d_shift_reg & 0x7FFFFFFF) ^ (CHIP_MAPPING[0] & 0x7FFFFFFF)) <= d_threshold) {
561                 if (d_pack_debug)
562                     fprintf(stderr, "Found %d 0 in chip sequence\n", d_preamble_cnt), fflush(stderr);
563                 // we found an other 0 in the chip sequence
564                 d_packet_byte = 0;
565                 d_preamble_cnt++;
566             } else if (gr::blocks::count_bits32((d_shift_reg & 0x7FFFFFFF) ^ (CHIP_MAPPING[7] & 0x7FFFFFFF)) <=
                    d_threshold) {
567                 if (d_pack_debug)
568                     fprintf(stderr, "Found first SFD\n"), fflush(stderr);
569                 d_packet_byte = 7 << 4;
570
571             } else {
572                 // we are not in the synchronization header
573                 if (d_pack_debug)
574                     fprintf(stderr, "Wrong first byte of SFD. %u\n", d_shift_reg), fflush(stderr);
575                 enter_search();
576                 break;
577             }
578
579         } else {
580             if (gr::blocks::count_bits32((d_shift_reg & 0x7FFFFFFF) ^ (CHIP_MAPPING[10] & 0x7FFFFFFF)) <= d_threshold) {
581                 d_packet_byte |= 0xA;
582                 if (d_pack_debug)
583                     fprintf(stderr, "Found sync, 0x%x\n", d_packet_byte), fflush(stderr);
584                 // found SFD
585                 // setup for header decode
586
587                 // each count number will be used to find
588                 // a relation between the usrp preambles
589                 if(d_debug)
590                 {
591                     preamble_marker.push_back(count);
592                 }
593

```

```

594         // if there's any data in from the previous run it will be appended as
595         // chances are that it will contain the preamble with the current data,
596         // only the previous input will be passed through as a safety precaution
597         // if the previously collected buffer doesn't exist then only the current
598         // data will be passed with the possibility of not receiving the preamble
599         if(d_prev_usrp_collect_buffer.is_activated()){
600             d_prev_buffer_activated = true;
601             d_current_preamble_buffer.append_buffer_data(d_prev_usrp_collect_buffer);
602             for(int c = 0; c < ninput_rx; c++)
603             {
604                 d_current_preamble_buffer.append_buffer_data(in_rx[c]);
605             }
606
607         }else{
608             d_prev_buffer_activated = false;
609             d_current_preamble_buffer.store_preamble_data(ninput_rx, in_rx);
610         }
611
612         enter_have_sync();
613         break;
614     } else {
615         if (d_chip_debug)
616             fprintf(stderr, "Wrong second byte of SFD. %u\n", d_shift_reg), fflush(stderr);
617         enter_search();
618         break;
619     }
620 }
621 }
622 }
623 }
624 break;
625
626 case STATE_HAVE_SYNC:
627     if (d_pack_debug)
628         fprintf(stderr, "Header Search bitcnt=%d, header=0x%08x\n", d_headerbitlen_cnt, d_header),
629         fflush(stderr);
630
631     while (count < ninput) {        // Decode the bytes one after another.
632         if (slice(inbuf[count++]))
633             d_shift_reg = (d_shift_reg << 1) | 1;
634         else
635             d_shift_reg = d_shift_reg << 1;
636
637         d_chip_cnt = d_chip_cnt+1;
638
639         if (d_chip_cnt == 32){
640             d_chip_cnt = 0;
641             unsigned char c = decode_chips(d_shift_reg);
642             if (c == 0xFF){
643                 // something is wrong. restart the search for a sync
644                 if (d_pack_debug)
645                     fprintf(stderr, "Found a not valid chip sequence! %u\n", d_shift_reg), fflush(stderr);
646
647                 enter_search();

```

```

648         break;
649     }
650
651     if (d_packet_byte_index == 0){
652         d_packet_byte = c;
653     } else {
654         // c is always < 15
655         d_packet_byte |= c << 4;
656     }
657     d_packet_byte_index = d_packet_byte_index + 1;
658     if (d_packet_byte_index%2 == 0){
659         // we have a complete byte which represents the frame length.
660         int frame_len = d_packet_byte;
661         if (frame_len <= MAX_PKT_LEN){
662
663             // keep track of the found sync markers
664             // to see correlate to good usrp markers
665             if (d_debug)
666             {
667                 sync_marker.push_back(count);
668             }
669
670             enter_have_header(frame_len);
671         } else {
672             enter_search();
673         }
674         break;
675     }
676 }
677 }
678 break;
679
680 case STATE_HAVE_HEADER:
681     if (d_pack_debug)
682         fprintf(stderr, "Packet Build count=%d, ninput=%d, packet_len=%d\n", count, ninput, d_packetlen), fflush(stderr);
683
684     while (count < ninput) { // shift bits into bytes of packet one at a time
685         if (slice(inbuf[count++]))
686             d_shift_reg = (d_shift_reg << 1) | 1;
687         else
688             d_shift_reg = d_shift_reg << 1;
689
690         d_chip_cnt = (d_chip_cnt+1)%32;
691
692         if (d_chip_cnt == 0){
693             unsigned char c = decode_chips(d_shift_reg);
694             if (c == 0xff){
695                 // something is wrong. restart the search for a sync
696                 if (d_pack_debug)
697                     fprintf(stderr, "Found a not valid chip sequence! %u\n", d_shift_reg), fflush(stderr);
698
699                 enter_search();
700                 break;
701             }

```

```

702
703 // the first symbol represents the first part of the byte.
704 if(d_packet_byte_index == 0){
705     d_packet_byte = c;
706 } else {
707     // c is always < 15
708     d_packet_byte |= c << 4;
709 }
710 //fprintf(stderr, "%d: 0x%x\n", d_packet_byte_index, c);
711 d_packet_byte_index = d_packet_byte_index + 1;
712 if(d_packet_byte_index%2 == 0){
713     // we have a complete byte
714     if (d_pack_debug)
715         fprintf(stderr, "packetcnt: %d, payloadcnt: %d, payload 0x%x, d_packet_byte_index: %d\n", d_packetlen_cnt,
716             d_payload_cnt, d_packet_byte, d_packet_byte_index), fflush(stderr);
717
718     d_packet[d_packetlen_cnt++] = d_packet_byte;
719     d_payload_cnt++;
720     d_packet_byte_index = 0;
721
722     if (d_payload_cnt >= d_packetlen){ // packet is filled, including CRC. might do check later in here
723         unsigned int scaled_lqi = (d_lqi / MAX_LQI_SAMPLES) << 3;
724         unsigned char lqi = (scaled_lqi >= 256? 255 : scaled_lqi);
725
726         pmt::pmt_t meta = pmt::make_dict();
727         meta = pmt::dict_add(meta, pmt::mp("lqi"), pmt::from_long(lqi));
728
729         std::memcpy(buf, d_packet, d_packetlen_cnt);
730         pmt::pmt_t payload = pmt::make_blob(buf, d_packetlen_cnt);
731
732         //***** Pass and Analyze Preamble Data Here!!!!*****
733         // the header count will be used as a way to
734         // correlate the header count with the USRP
735         // data
736         if (d_debug)
737         {
738             header_marker.push_back(count);
739         }
740
741         std::tuple<dna_buffer, bool> buffer_data;
742         if (d_prev_buffer_activated)
743         {
744             buffer_data = std::move(normalize_preamble(d_current_preamble_buffer, ninput_rx, d_burst_trigger));
745         }
746
747         d_preamble_processed = std::get<1>(buffer_data);
748
749         //based on the last run, it will determine if a
750         //preamble met the criteria to be sent or not
751         if (d_preamble_processed == true)
752         {
753             d_sent_preambles++;
754         } else {

```

```

755         d_removed_preambles++;
756     }
757
758     // raw data crc will be checked for validity and filtering
759     // later on
760     uint16_t crc_check = crc16(buf, d_packetlen_cnt);
761
762     // if-else statements are a way to keep track of processing errors
763     // of the general_work function based on mac_in
764     // function from mac.cc from gr-ieee802.15.4
765     d_recent_packet_length = d_packetlen;
766     if(d_packetlen_cnt < 9)
767     {
768         d_short_payload++;
769     }else if(crc_check)
770     {
771         d_crc_bad_payload++;
772     }else
773     {
774         d_crc_good_payload++;
775     }
776
777     // the if/else conditions will check to see if the user will
778     // accept only properly sized crc verified packet outputs or
779     // any and everything that comes in packet-wise
780     if(d_crc_set)
781     {
782         if(!crc_check && (d_packetlen_cnt < 9))
783         {
784             // might make a function later on to handle empty values
785             // but a loop will suffice for now
786             for(int c = 0; c < ninput_rx; c++)
787             {
788                 d_crc_preamble_buffer[c] = gr_complex(0.0f, 0.0f);
789             }
790
791         }else
792         {
793             d_crc_preamble_buffer = d_prev_buffer_activated ? std::move(std::get<0>(buffer_data)) : std::move(
                d_current_preamble_buffer);
794         }
795     }else
796     {
797         d_crc_preamble_buffer = d_prev_buffer_activated ? std::move(std::get<0>(buffer_data)) : std::move(
                d_current_preamble_buffer);
798     }
799
800     //debug check copied directly from mac.cc
801     //from gr-ieee802.15.4
802     if(d_usrp_debug)
803     {
804         if(d_packetlen_cnt < 9)
805         {
806             fprintf(stderr, "\nDetected packet MAC frame too short"), fflush(stderr);

```

```

807         }
808         if (crc_check)
809         {
810             fprintf(stderr, "\nDetected packet MAC failed CRC check"), fflush(stderr);
811         } else
812         {
813             fprintf(stderr, "\nDetected packet MAC passed CRC check\n"), fflush(stderr);
814         }
815     }
816
817     if (d_save_stats_switch == SAVE::YES)
818     {
819         clear_stream();
820         d_msg_stream << "===== " <<
            "\n";
821         d_msg_stream << "Current Burst Sample Length: " << ninput_rx << "\n";
822         if (d_prev_buffer_activated)
823         {
824             d_msg_stream << "Buffer + Detected Burst Sample length: ";
825         } else
826         {
827             d_msg_stream << "Detected Burst Sample: ";
828         }
829         d_msg_stream << d_current_preamble_buffer.get_sample_length() << "\n";
830         d_msg_stream << "Total Packets Detected: " << d_short_payload + d_crc_bad_payload + d_crc_good_payload << "\n";
831         d_msg_stream << "Short Packets: " << d_short_payload << "\n";
832         d_msg_stream << "Bad CRC Packets: " << d_crc_bad_payload << "\n";
833         d_msg_stream << "Good CRC Packets: " << d_crc_good_payload << "\n";
834         d_msg_stream << "Total Preambles Processed" << "\n";
835         d_msg_stream << "of " << d_short_payload + d_crc_bad_payload + d_crc_good_payload << " Total Packets Detected: ";
836         d_msg_stream << d_removed_preambles + d_sent_preambles << "\n";
837         d_msg_stream << "Preambles Accepted: " << d_sent_preambles << "\n";
838         d_msg_stream << "Preambles Rejected: " << d_removed_preambles << "\n";
839         d_msg_stream << "===== " <<
            "\n";
840         d_stats_writer->store_data(d_msg_stream.str().size(), (void*)d_msg_stream.str().c_str());
841     }
842
843     message_port_pub(pmt::mp("out"), pmt::cons(meta, payload));
844
845     if (d_pack_debug)
846         fprintf(stderr, "Adding message of size %d to queue\n", d_packetlen_cnt);
847     enter_search();
848     break;
849 }
850 }
851 }
852 }
853 break;
854
855 default:
856     assert(0);

```

```

857         break;
858
859     }
860 }
861
862
863 if(d_pack_debug)
864     fprintf(stderr, "Samples Processed: %d\n", ninput_items[0]), fflush(stderr);
865
866     consume(0, ninput_items[0]);
867
868 // Debug & Iteration Statistics for the blocks current run
869 if(d_usrp_debug && (d_crc_preamble_buffer.is_activated()))
870 {
871     fprintf(stderr,"++++++++++++++++++++++++++++++++++++"), fflush(stderr);
872     fprintf(stderr, "\nM & M Timer Input Count: %d\tUSRP Input Count: %d",
873             ninput, ninput_rx), fflush(stderr);
874     fprintf(stderr, "\nInherited Output Item Count: %d", noutput_items), fflush(stderr);
875     fprintf(stderr, "\nPacket Length: %d", d_recent_packet_length), fflush(stderr);
876     fprintf(stderr, "\nTotal Processed Preambles: %d", (d_removed_preambles +
877             d_sent_preambles)), fflush(stderr);
878     fprintf(stderr, "\nDropped Preambles: %d", d_removed_preambles), fflush(stderr);
879     fprintf(stderr, "\nSent Preambles: %d", d_sent_preambles), fflush(stderr);
880
881     vector_print("\nPreamble", preamble_marker);
882     vector_print("Sync", sync_marker);
883     vector_print("Header", header_marker);
884
885     fprintf(stderr, "\nTotal Run Numbers => Short Packets: %d\tBad CRC Packets: %d\tGood CRC Packets: %d\n",
886             d_short_payload, d_crc_bad_payload, d_crc_good_payload), fflush(stderr);
887
888     preamble_marker.clear();
889     sync_marker.clear();
890     header_marker.clear();
891     fprintf(stderr,"-----"), fflush(stderr);
892 }
893
894 // The current input will be appended here to the FIFO buffer sturcture for the
895 // next iteration.
896 d_prev_usrp_collect_buffer.append_burst_sample_size(ninput_rx);
897
898 for(int c = 0; c < ninput_rx; c++ )
899 {
900     out_rx[c] = d_crc_preamble_buffer.is_activated() ? d_crc_preamble_buffer[c] : gr_complex(0.0F,0.0F);
901     d_prev_usrp_collect_buffer.append_buffer_data(in_rx[c]);
902     out_usrp[c] = in_rx[c];
903 }
904 d_crc_preamble_buffer.unset_buffer();
905
906 consume(0, 0);
907 consume(1, ninput_rx);
908
909 return ninput_rx;
910 }

```

```

911
912 void
913 preamble_sink_impl::enter_search()
914 {
915     if (d_chip_debug)
916     {
917         fprintf(stderr, "@ enter_search\n");
918     }
919
920     d_state = STATE_SYNC_SEARCH;
921     d_shift_reg = 0;
922     d_preamble_cnt = 0;
923     d_chip_cnt = 0;
924     d_packet_byte = 0;
925     d_current_preamble_buffer.unset_buffer();
926 }
927
928 void
929 preamble_sink_impl::enter_have_sync()
930 {
931     if (d_chip_debug)
932     {
933         fprintf(stderr, "@ enter_have_sync\n");
934     }
935
936     d_state = STATE_HAVE_SYNC;
937     d_packetlen_cnt = 0;
938     d_packet_byte = 0;
939     d_packet_byte_index = 0;
940
941     // Link Quality Information
942     d_lqi = 0;
943     d_lqi_sample_count = 0;
944 }
945
946 void
947 preamble_sink_impl::enter_have_header(const int &payload_len)
948 {
949     if (d_chip_debug)
950         fprintf(stderr, "@ enter_have_header (payload_len = %d)\n", payload_len);
951
952     d_state = STATE_HAVE_HEADER;
953     d_packetlen = payload_len;
954     d_payload_cnt = 0;
955     d_packet_byte = 0;
956     d_packet_byte_index = 0;
957 }
958
959 unsigned char
960 preamble_sink_impl::decode_chips(const unsigned int &chips)
961 {
962     int i;
963     int best_match = 0xFF;
964     int min_threshold = 33; // Matching to 32 chips, could never have a error of 33 chips

```

```

965
966     for(i=0; i<16; i++)
967     {
968         // FIXME: we can store the last chip
969         // ignore the first and last chip since it depends on the last chip.
970         unsigned int threshold = gr::blocks::count_bits32((chips & 0x7FFFFFFE) ^ (CHIP_MAPPING[i] & 0x7FFFFFFE));
971
972         if (threshold < min_threshold)
973         {
974             best_match = i;
975             min_threshold = threshold;
976         }
977     }
978
979     if (min_threshold < d_threshold) {
980         if (d_chip_debug)
981         {
982             fprintf(stderr, "Found sequence with %d errors at 0x%x\n", min_threshold, (chips & 0x7FFFFFFE) ^ (CHIP_MAPPING[
983                 best_match] & 0x7FFFFFFE)), fflush(stderr);
984         }
985
986         // LQI: Average number of chips correct * MAX_LQI_SAMPLES
987         //
988         if (d_lqi_sample_count < MAX_LQI_SAMPLES)
989         {
990             d_lqi += 32 - min_threshold;
991             d_lqi_sample_count++;
992         }
993
994         return std::move((char)best_match & 0xF);
995     }
996
997     return std::move(0xFF);
998 }
999
1000 int
1001 preamble_sink_impl::slice(const float &x)
1002 {
1003     return std::move(x > 0 ? 1 : 0);
1004 }
1005
1006 void
1007 preamble_sink_impl::set_threshold(const unsigned int &threshold)
1008 {
1009     d_threshold = threshold;
1010 }
1011
1012 unsigned int
1013 preamble_sink_impl::get_threshold() const
1014 {
1015     return d_threshold;
1016 }
1017
1018 void

```

```

1018     preamble_sink_impl::set_samp_rate(const float &samp_rate)
1019     {
1020         d_samp_rate = samp_rate;
1021         d_phy_802_15_4.set_sample_rate(samp_rate);
1022     }
1023
1024     void
1025     preamble_sink_impl::set_buffer_size(const unsigned int &buffer_size)
1026     {
1027         if (buffer_size <=0)
1028         {
1029             std::cout << "\nBuffer size too small...It's now set to 1" << std::endl;
1030             d_buffer_size = 1;
1031         } else
1032         {
1033             d_buffer_size = buffer_size;
1034         }
1035     }
1036
1037     unsigned int
1038     preamble_sink_impl::get_buffer_size() const
1039     {
1040         return d_buffer_size;
1041     }
1042
1043     float
1044     preamble_sink_impl::get_samp_rate() const
1045     {
1046         return d_samp_rate;
1047     }
1048
1049     void
1050     preamble_sink_impl::set_burst_trigger(const float &burst_trigger)
1051     {
1052         d_burst_trigger = burst_trigger;
1053     }
1054
1055     float
1056     preamble_sink_impl::get_burst_trigger() const
1057     {
1058         return d_burst_trigger;
1059     }
1060
1061     void
1062     preamble_sink_impl::set_debug(const DebugType &debug)
1063     {
1064         d_debug = debug;
1065
1066         // quick bitwise comparison will be used to determine
1067         // what level of debuggin will be established
1068         d_chip_debug = (debug & CHIP_ON) > 0 ? 1 : 0;
1069         d_pack_debug = (debug & PACKET_ON) > 0 ? 1 : 0;
1070         d_usrp_debug = (debug & USRP_ON) > 0 ? 1 : 0;
1071     }

```

```

1072
1073     DebugType
1074     preamble_sink_impl::get_debug() const
1075     {
1076         return d_debug;
1077     }
1078
1079     void
1080     preamble_sink_impl::set_crc(const CRC &crc)
1081     {
1082         d_crc = crc;
1083
1084         d_crc_set = (crc & CRC_ONLY) > 0 ? 1 : 0;
1085     }
1086
1087     void
1088     preamble_sink_impl::set_save_stats_switch(const SAVE &save_stats_switch)
1089     {
1090         d_save_stats_switch = save_stats_switch;
1091
1092         d_save_switch_set = (save_stats_switch > 0) ? 1 : 0;
1093     }
1094
1095     void
1096     preamble_sink_impl::set_save_stats_file(const std::string &save_stats_file)
1097     {
1098         d_save_stats_file = save_stats_file;
1099     }
1100
1101     SAVE
1102     preamble_sink_impl::get_save_stats_switch() const
1103     {
1104         return d_save_stats_switch;
1105     }
1106
1107     std::string
1108     preamble_sink_impl::get_save_stats_file() const
1109     {
1110         return d_save_stats_file;
1111     }
1112
1113
1114     CRC
1115     preamble_sink_impl::get_crc() const
1116     {
1117         return d_crc;
1118     }
1119
1120     uint16_t
1121     preamble_sink_impl::crc16(const char *buf, const int &len)
1122     {
1123         uint16_t crc = 0;
1124
1125         for(int i = 0; i < len; i++) {

```

```

1126         for(int k = 0; k < 8; k++) {
1127             int input_bit = (!!(buf[i] & (1 << k)) ^ (crc & 1));
1128             crc = crc >> 1;
1129             if(input_bit) {
1130                 crc ^= (1 << 15);
1131                 crc ^= (1 << 10);
1132                 crc ^= (1 << 3);
1133             }
1134         }
1135     }
1136
1137     return std::move(crc);
1138 }
1139
1140 void
1141 preamble_sink_impl::clear_stream()
1142 {
1143     d_msg_stream.str(std::string());
1144 }
1145
1146 void
1147 preamble_sink_impl::vector_print(const std::string &label,
1148                                 const std::vector<int> &input)
1149 {
1150     if(input.empty())
1151     {
1152         fprintf(stderr, "No %s Detected\n", label.c_str()), fflush(stderr);
1153     }else
1154     {
1155         fprintf(stderr, "%s Detected at : ", label.c_str());
1156         for(auto i : input)
1157         {
1158             fprintf(stderr, "%d ", i);
1159         }
1160         fprintf(stderr, "\n"), fflush(stderr);
1161     }
1162 }
1163
1164 std::tuple<dna_buffer, bool>
1165 preamble_sink_impl::normalize_preamble(const dna_buffer &buffer,
1166                                       const int &max_return_size,
1167                                       const float &burst_trigger)
1168 {
1169     dna_buffer position_adjusted_vector;
1170     float absolute_magnitude = 0.0f;
1171     int absolute_magnitude_location = 0;
1172     int burst_trigger_location = 0;
1173     float temp_magnitude = 0;
1174     bool preamble_processed = false;
1175
1176     /*****
1177     Add Proper filter_burst Burst Detector Down the Road!!!!
1178     *****/
1179

```

```

1180 // first step is to go through and find the min/max of the burst
1181 for(int c = 0; c < buffer.get_sample_length(); c++)
1182 {
1183     temp_magnitude = std::fabs(buffer[c]);
1184     if(temp_magnitude > absolute_magnitude)
1185     {
1186         absolute_magnitude = temp_magnitude;
1187         absolute_magnitude_location = c;
1188     }
1189 }
1190
1191 // then we divide by the largest magnitude to normalize the burst
1192 // if the signal is pure noise, then it's going to return a burst of
1193 // pure noise
1194 bool trigger_set = false;
1195 for(int c = 0; c < buffer.get_sample_length(); c++)
1196 {
1197     temp_magnitude = std::fabs(buffer[c]);
1198     position_adjusted_vector.append_buffer_data(buffer[c]);
1199
1200     //need a flag to ensure loop doesn't keep modifying burst point
1201     if((temp_magnitude/absolute_magnitude >= burst_trigger) && (trigger_set == false))
1202     {
1203         trigger_set = true;
1204         burst_trigger_location = c;
1205     }
1206 }
1207
1208 int buffer_stop_element = d_phy_802_15_4.get_preamble_len_samples() + // 640 samples for SHR @4MS/s
1209     burst_trigger_location;
1210
1211 int zero_buffer_length = 0;
1212 if((buffer_stop_element + 1) > position_adjusted_vector.get_sample_length())
1213 {
1214     //case where the preamble is not available because it's index is beyond the
1215     //current buffers max
1216     position_adjusted_vector.unset_buffer();
1217     position_adjusted_vector.insert_rear_spacing(max_return_size);
1218 }
1219 }else{
1220     // case where we have more than enough samples to capture the preamble
1221
1222     // first we're going to remove the elements beyond the preamble location, then
1223     // we're going to remove the front where it will most likely be noise
1224     position_adjusted_vector.erase_buffer_range(buffer_stop_element, position_adjusted_vector.get_sample_length() - 1);
1225     position_adjusted_vector.erase_buffer_range(0, burst_trigger_location - 1);
1226
1227     // Afterwards we will set the remainder of the input vector to zero and returned
1228     // the normalized and non-normalized shifted preamble data
1229     zero_buffer_length = max_return_size - position_adjusted_vector.get_sample_length();
1230     position_adjusted_vector.insert_rear_spacing(zero_buffer_length);
1231     preamble_processed = true;
1232 }
1233

```

```

1234     return std::move(std::make_tuple(position_adjusted_vector, preamble_processed));
1235 }
1236
1237 std::tuple<dna_buffer,int>
1238 preamble_sink_impl::filter_burst(const dna_buffer & buffer, const int &filter_taps)
1239 {
1240     // First, we need to generate the averaging filter taps by creating a vector
1241     // that's the length of the input of filter taps with each tap weighted 1/filter_taps
1242     std::vector<float> filter;
1243     for(int c = 0; c < filter_taps; c++)
1244     {
1245         filter.push_back(1.0f/filter_taps);
1246     }
1247
1248     // these two values will be used to keep track
1249     // of the maximum magnitude while the signal is being filtered
1250     float absolute_max_value = 0.0f;
1251     float temp_max_value = 0.0f;
1252     int absolute_max_value_position = 0;
1253
1254     // the length of a discrete signal after convolution is the sum of the two signals less
1255     // one element
1256     std::vector<float> filtered_burst;
1257     float burst_sum = 0.0f;
1258
1259     for(int i = 0; i < buffer.get_sample_length(); i++)
1260     {
1261         burst_sum = 0.0f;
1262         for(int j = 0; j < filter_taps; j++)
1263         {
1264             if(((i - j) > 0) && (i >= (filter_taps - 1)))
1265             {
1266                 burst_sum += std::fabs(buffer[i - j])*filter[j];
1267             }
1268         }
1269     }
1270     filtered_burst.push_back(burst_sum);
1271 }
1272
1273 }

```

D.1.2 Packet Sink XML GRC Code.

```

1 <?xml version="1.0"?>
2 <block>
3   <name>preamble_sink</name>
4   <key>ieee802_15_4_preamble_sink</key>
5   <category>[ieee802_15_4]</category>
6   <import>import ieee802_15_4</import>
7   <make>ieee802_15_4.preamble_sink($samp_rate, $threshold, $buffer, $burst_trigger,
8                                   $debug, $src, $save_switch, $save_file)</make>
9   <param>

```

```
10     <name>Samp_rate</name>
11     <key>samp_rate</key>
12     <type>float</type>
13 </param>
14 <param>
15     <name>Chip_Error_Threshold_Max</name>
16     <key>threshold</key>
17     <type>int</type>
18 </param>
19
20 <param>
21     <name>Buffer_Length</name>
22     <key>buffer</key>
23     <type>int</type>
24 </param>
25
26 <param>
27     <name>Burst_Trigger_Percentage</name>
28     <key>burst_trigger</key>
29     <type>float</type>
30 </param>
31 <param>
32     <name>CRC</name>
33     <key>crc</key>
34     <type>enum</type>
35     <option>
36         <name>ALL_DATA</name>
37         <key>0</key>
38     </option>
39     <option>
40         <name>CRC_ONLY</name>
41         <key>1</key>
42     </option>
43 </param>
44 <param>
45     <name>Save_Stats?</name>
46     <key>save_switch</key>
47     <type>enum</type>
48     <option>
49         <name>NO</name>
50         <key>0</key>
51     </option>
52     <option>
53         <name>YES</name>
54         <key>1</key>
55     </option>
56 </param>
57 <param>
58     <name>Default_Stats_Name/Location</name>
59     <key>save_file</key>
60     <type>file_save</type>
61 </param>
62 <param>
63     <name>Debug</name>
```

```

64     <key>debug</key>
65     <type>enum</type>
66     <option>
67         <name>ALL_OFF</name>
68         <key>0</key>
69     </option>
70     <option>
71         <name>CHIP_ON</name>
72         <key>1</key>
73     </option>
74     <option>
75         <name>PACKET_ON</name>
76         <key>2</key>
77     </option>
78     <option>
79         <name>USRP_ON</name>
80         <key>4</key>
81     </option>
82     <option>
83         <name>CHIP_PACKET_ON</name>
84         <key>3</key>
85     </option>
86     <option>
87         <name>CHIP_USRP_ON</name>
88         <key>5</key>
89     </option>
90     <option>
91         <name>PACKET_USRP_ON</name>
92         <key>6</key>
93     </option>
94     <option>
95         <name>ALL_ON</name>
96         <key>7</key>
97     </option>
98 </param>
99 <sink>
100     <name>in</name>
101     <type>float</type>
102 </sink>
103 <sink>
104     <name>in</name>
105     <type>complex</type>
106 </sink>
107 <source>
108     <name>out</name>
109     <type>message</type>
110 </source>
111 <source>
112     <name>preamble_out</name>
113     <type>complex</type>
114 </source>
115 <source>
116     <name>usrp_passthrough</name>
117     <type>complex</type>

```

118 </source>
119 </block>

D.2 Dna detector ccf Block Source Code

D.2.1 Dna detector ccf C++ Code.

```
1  /*****
   ***
2  *preamble_sink class
3  *****/
   **/
4  /* -- c++ -- */
5  /*
6   * Copyright 2018 gr-ieee802_15_4 author.
7   *
8   * This is free software; you can redistribute it and/or modify
9   * it under the terms of the GNU General Public License as published by
10  * the Free Software Foundation; either version 3, or (at your option)
11  * any later version.
12  *
13  * This software is distributed in the hope that it will be useful,
14  * but WITHOUT ANY WARRANTY; without even the implied warranty of
15  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
16  * GNU General Public License for more details.
17  *
18  * You should have received a copy of the GNU General Public License
19  * along with this software; see the file COPYING. If not, write to
20  * the Free Software Foundation, Inc., 51 Franklin Street,
21  * Boston, MA 02110-1301, USA.
22  */
23
24
25 #ifndef INCLUDED_IEEE802_15_4_DNA_DETECTOR_CCF_H
26 #define INCLUDED_IEEE802_15_4_DNA_DETECTOR_CCF_H
27
28 #include <ieee802_15_4/api.h>
29 #include <gnuradio/block.h>
30 #include <string>
31 #include "preamble_sink.h"
32
33 namespace gr {
34     namespace ieee802_15_4 {
35         /*!
36          * SWITCH is a simple on/off enum option
37          */
38         enum SWITCH{
39             OFF = 0,
40             ON = 1
41         };
```

```

42  /*!
43  * Class: dna_detector
44  * Author: Cruz, Frankie
45  * Description: the class is calculates which emitter is being received
46  *               by performing rf-dna analysis and outputting the results
47  *               to a histogram via msg passing and reduces the ammount
48  *               of output data bursts passed in by reducing the "dead"
49  *               space in between
50  * Generated: Mon Dec 3 17:30:12 2018
51  * Modified: Sat Feb 2 9:00:28 2018
52  * Build Version: 1.11
53  * Changes: -Cleaned code up a bit
54  */
55  class IEEE802_15_4_API dna_detector_ccf : virtual public gr::block
56  {
57  public:
58      typedef boost::shared_ptr<dna_detector_ccf> sptr;
59      /*!
60      * param fingerprints is the number of emitters that will be analyzed
61      * param dna_row_size is the row size of the imported rfdna fingerprints
62      * param dna_column_size is the column size of the imported rfdna fingerprints
63      * param dna_region_num is the number of regions used for rfdna fingerprints
64      * param samp_rate is the sample rate of the system
65      * param fingerprint_filename is the filename the user picks to store fingerprints
66      * param fingerprint_save_switch is the on/off to store fingerprints
67      * param class_matrix_filename is the trained classification file to load into a matrix
68      * param means_matrix_filename is the trained device means file to load into a matrix
69      * param class_matrix_load_switch is the on/off to load the DNA file
70      * param norm_vector_filename is the scaling multiplier for fingerprints generated by trained data
71      * param xoffset_vector_filename is an offset that has to be applied to the generated fingerprints
72      * param save_stats_switch is the on/off to store running stats info (note: write can be behind)
73      * param save_dna_results is the on/off to store the result of the fingerprint and DNA matrix
74      */
75      static sptr make(int devices, int dna_row_size, int dna_col_size, int dna_region_num,
76                      DebugType debug, float samp_rate, std::string fingerprint_filename,
77                      SWITCH fingerprint_save_switch, std::string class_matrix_filename,
78                      std::string means_matrix_filename, std::string norm_vector_filename,
79                      std::string xoffset_vector_filename, SWITCH class_matrix_load_switch,
80                      SWITCH save_stats_switch, SWITCH save_dna_results);
81
82      virtual void set_devices(const int &devices) = 0;
83      virtual void set_dna_row_size(const int &row_size) = 0;
84      virtual void set_dna_column_size(const int &col_size) = 0;
85      virtual void set_dna_region_num(const int &region_num) = 0;
86      virtual void set_debug(const DebugType &debug) = 0;
87      virtual void set_dna_fingerprint_filename(const std::string &fingerprint_filename) = 0;
88      virtual void set_dna_fingerprint_save_switch(const SWITCH &fingerprint_save_switch) = 0;
89      virtual void set_dna_classification_matrix_filename(const std::string &class_matrix_filename) = 0;
90      virtual void set_dna_means_matrix_filename(const std::string &means_matrix_filename) = 0;
91      virtual void set_dna_classification_load_switch(const SWITCH &class_matrix_load_switch) = 0;
92      virtual void set_save_stats_switch(const SWITCH &save_stats_switch) = 0;
93      virtual void set_save_dna_results_switch(const SWITCH &save_dna_results_switch) = 0;
94      virtual void set_samp_rate(const float &samp_rate) = 0;
95      virtual void set_dna_norm_vector_filename(const std::string &norm_vector_filename) = 0;

```

```

96     virtual void set_dna_xoffset_vector_filename(const std::string &xoffset_vector_filename) = 0;
97
98     virtual int get_devices() const = 0;
99     virtual int get_dna_row_size() const = 0;
100    virtual int get_dna_column_size() const = 0;
101    virtual int get_dna_region_num() const = 0;
102    virtual DebugType get_debug() const = 0;
103    virtual std::string get_dna_fingerprint_filename() const = 0;
104    virtual SWITCH get_dna_fingerprint_save_switch() const = 0;
105    virtual std::string get_dna_classification_matrix_filename() const = 0;
106    virtual std::string get_dna_means_matrix_filename() const = 0;
107    virtual SWITCH get_dna_classification_load_switch() const = 0;
108    virtual SWITCH get_save_stats_switch() const = 0;
109    virtual SWITCH get_save_dna_results_switch() const = 0;
110    virtual float get_samp_rate() const = 0;
111    virtual std::string get_dna_norm_vector_filename() const = 0;
112    virtual std::string get_dna_xoffset_vector_filename() const = 0;
113 };
114
115 } // namespace ieee802_15_4
116 } // namespace gr
117 #endif /* INCLUDED_IEEE802_15_4_DNA_DETECTOR_CCF_H */
118
119 /*****
120  *preamble_sink_impl.h implemtation class definitions
121  *****/
122 /**/
123 /* --* c++ -* */
124 /*
125  * Copyright 2018 gr-ieee802_15_4 author.
126  *
127  * This is free software; you can redistribute it and/or modify
128  * it under the terms of the GNU General Public License as published by
129  * the Free Software Foundation; either version 3, or (at your option)
130  * any later version.
131  *
132  * This software is distributed in the hope that it will be useful,
133  * but WITHOUT ANY WARRANTY; without even the implied warranty of
134  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
135  * GNU General Public License for more details.
136  *
137  * You should have received a copy of the GNU General Public License
138  * along with this software; see the file COPYING. If not, write to
139  * the Free Software Foundation, Inc., 51 Franklin Street,
140  * Boston, MA 02110-1301, USA.
141  */
142 #ifndef INCLUDED_IEEE802_15_4_DNA_DETECTOR_CCF_IMPL_H
143 #define INCLUDED_IEEE802_15_4_DNA_DETECTOR_CCF_IMPL_H
144
145 #include <ieee802_15_4/dna_detector_ccf.h>
146 #include "preamble_sink_impl.h" //included to add class types stored i.e. dna_buffer packets
147

```

```

148 // Boost matrix libraries
149 #include <boost/numeric/ublas/vector.hpp>
150 #include <boost/numeric/ublas/matrix.hpp>
151 #include <boost/numeric/ublas/operation.hpp>
152 #include <boost/numeric/ublas/operations.hpp>
153 #include <boost/numeric/ublas/io.hpp>
154
155 // Boost Stats Libraries
156 #include <boost/math/distributions.hpp>
157 #include <boost/bind.hpp>
158 #include <boost/ref.hpp>
159 #include <boost/accumulators/accumulators.hpp>
160 #include <boost/accumulators/statistics/stats.hpp>
161 #include <boost/accumulators/statistics/mean.hpp>
162 #include <boost/accumulators/statistics/moment.hpp>
163 #include <boost/accumulators/statistics/variance.hpp>
164 #include <boost/accumulators/statistics/skewness.hpp>
165 #include <boost/accumulators/statistics/kurtosis.hpp>
166
167 // Eigen Matrix libraries
168 #include <eigen3/Eigen/Dense>
169
170 //std libraries
171 #include <sstream>
172 #include <iostream>
173 #include <fstream>
174 #include <memory>
175 #include <cmath>
176 #include <iomanip>
177 #include <ctime>
178
179 namespace gr {
180     namespace ieee802_15_4 {
181
182         //shorthand syntax simplifier for calling Eignen complex matricies and vectors
183         template<class T>
184             using EigMat = Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>; //EigRow is a N x 1 matrix shorthand
185         template<class T>
186             using EigColVec = Eigen::Matrix<T, Eigen::Dynamic, 1>; //EigColVec is a N x 1 vector shorthand
187         template<class T>
188             using EigRowVec = Eigen::Matrix<T, 1, Eigen::Dynamic>; //EigRowVec is a 1 x M vector shorthand
189
190         //shorthand syntax for calling boost specific matricies and vectors
191         template<class V>
192             using BoostMat = boost::numeric::ublas::matrix<V>; //specifiy N x M at generation
193         template<class V>
194             using BoostVec = boost::numeric::ublas::vector<V>; //specify vector length at generation
195
196         // using the accumulator namespace and dna_marker
197         // alias provide some flexibility on the data being analyzed
198         // and shortening the ammount of code to type....
199         using namespace boost::accumulators; // shortening boost call
200         template<class X>

```

```

201 using dna_marker = boost::accumulators::accumulator_set<X, features<tag::count, tag::mean(immediate), tag::variance(
    immediate), tag::skewness, tag::kurtosis>>;
202
203 class dna_detector_ccf_impl : public dna_detector_ccf
204 {
205 private:
206     enum{SEARCHING, DETECTED, FINGERPRINT} d_state; // keep track of what state we're in
207
208     int d_devices = 5; // number of devices/fingerprints input for analysis
209     int d_usrp_debug = 0; // USRP verbose output for debugging
210     int d_dna_row_size; // dna fingerprints row dimension
211     int d_dna_column_size; // dna fingerprints column dimension
212     int d_dna_region_num = 10; // # of fingerprint regions
213     float d_samp_rate = 4e6; //default sample rate
214
215     std::stringstream d_msg_stream; // designed to handle messages string messages
216     std::stringstream d_emitter_stats_stream; // designed to save results info for each emmitters results
217     std::string d_fingerprint_filename; // filename to save fingerprints
218     std::string d_fingerprint_stats_filename; // filename to save stats
219     std::string d_matrix_mult_results_filename; // filename to save
220     std::string d_norm_vector_filename; // normalization vectors filename
221     std::string d_offset_vector_filename; // offset vectors filename
222     SWITCH d_fingerprint_save_switch = SWITCH::OFF; // option to save the fingerprints generated per run
223     std::string d_class_matrix_filename; // classification matrix file name
224     std::string d_means_matrix_filename; // means matrix file name
225     SWITCH d_class_matrix_load_switch = SWITCH::OFF; // option to load a classifier file
226     SWITCH d_save_stats_switch = SWITCH::OFF; // option to save the generated statistics
227     SWITCH d_save_dna_results_switch = SWITCH::OFF; // option to save dna multiplication results
228
229     //File Read & Write Modifiers
230     std::shared_ptr<file_read_impl> d_offset_vector_loader; // loades calculated fingerprint offsets
231     std::shared_ptr<file_read_impl> d_norm_vector_loader; // loads calculated normalization scaling vector
232     std::shared_ptr<file_read_impl> d_class_means_loader; // loads calculated means for each device generated by MATLAB
233     std::shared_ptr<file_read_impl> d_dna_loader; // loads classification matrix generated by MATLAB
234     std::shared_ptr<file_store_impl> d_dna_fingerprint_write; // for d_dna_fingerprints
235     std::shared_ptr<file_store_impl> d_stats_writer; //for statistis output file
236     std::shared_ptr<file_store_impl> d_matrix_results_writer; // stores results of multiplication matrix
237     std::shared_ptr<file_store_impl> d_dna_results_writer; //stores results of dna multiplication
238
239     DebugType d_debug = ALL_OFF; // debug type
240     unsigned long int d_burst_count = 0; // burst counter(4,294,967,295 size limit)
241     unsigned long int d_fingerprints_counter = 0; // fingerprint generation counter
242     unsigned long int d_nan_counter = 0; //keeps track of fingerprints that were not sent
243
244     ieee802_15_4_packet d_phy_802_15_4; //contains all the class sizing functions
245
246     std::vector<unsigned long int> d_emitter_bin_counter; // array keeping track of what devices were identified
247     std::vector<gr_complex> d_current_detected_preamble;
248     std::vector<float> d_dna_results;
249     std::vector<float> d_dna_fingerprints; // calculated fingerprints from detected preamble
250     EigMat<float> d_dna_matrix; // classification matrix generated from matlab
251     EigMat<float> d_class_means_matrix; // device classification means generated from matlab
252     EigRowVec<float> d_norm_vector; // normalization vector to scale incomming fingerprint by
253     EigRowVec<float> d_offset_vector; // offset vector to adjust collected fingerprints

```

```

254
255 public:
256     dna_detector_ccf_impl(int devices, int dna_row_size, int dna_column_size, int dna_region_num,
257                           DebugType debug, float samp_rate, std::string fingerprint_filename,
258                           SWITCH fingerprint_save_switch, std::string class_matrix_filename,
259                           std::string means_matrix_filename, std::string norm_vector_filename,
260                           std::string xoffset_vector_filename, SWITCH class_matrix_load_switch,
261                           SWITCH save_stats_switch, SWITCH save_dna_results_switch);
262
263 ~dna_detector_ccf_impl();
264
265 // Where all the action really happens
266 void forecast (int noutput_items, gr_vector_int &ninput_items_required);
267
268 int general_work(int noutput_items,
269                 gr_vector_int &ninput_items,
270                 gr_vector_const_void_star &input_items,
271                 gr_vector_void_star &output_items);
272
273 // state functions to process data
274 void burst_search();
275 void burst_detect();
276 void burst_fingerprint();
277
278 // publicly available setters
279 void set_devices(const int &devices); //set the # of devices
280 void set_dna_row_size(const int &dna_row_size); // set the row size of dna matrix, ie N of N x M
281 void set_dna_column_size(const int &dna_column_size); // set the column size of the dna matrix, ie M of N x M
282 void set_dna_region_num(const int &dna_region_num); // set the number of regions used in fingerprinting
283 void set_debug(const DebugType &debug);
284 void set_dna_fingerprint_filename(const std::string &fingerprint_filename);
285 void set_dna_fingerprint_save_switch(const SWITCH &fingerprint_save_switch);
286 void set_dna_classification_matrix_filename(const std::string &class_matrix_filename);
287 void set_dna_means_matrix_filename(const std::string &mean_matrix_filename);
288 void set_dna_classification_load_switch(const SWITCH &class_matrix_load_switch);
289 void set_save_stats_switch(const SWITCH &save_stats_switch);
290 void set_save_dna_results_switch(const SWITCH &save_dna_results_switch);
291 void set_samp_rate(const float &samp_rate);
292 void set_dna_norm_vector_filename(const std::string &norm_vector_filename);
293 void set_dna_xoffset_vector_filename(const std::string &xoffset_vector_filename);
294 void clear_stream(); // shorthand way to clear d_msg_stream via stackoverflow.com
295
296 //publicly available getters
297 int get_devices() const;
298 int get_dna_row_size() const;
299 int get_dna_column_size() const;
300 int get_dna_region_num() const;
301 DebugType get_debug() const;
302 std::string get_dna_fingerprint_filename() const;
303 SWITCH get_dna_fingerprint_save_switch() const;
304 std::string get_dna_classification_matrix_filename() const;
305 std::string get_dna_means_matrix_filename() const;
306 SWITCH get_dna_classification_load_switch() const;
307 SWITCH get_save_stats_switch() const;

```

```

308     SWITCH get_save_dna_results_switch() const;
309     float get_samp_rate() const;
310     std::string get_dna_norm_vector_filename() const;
311     std::string get_dna_xoffset_vector_filename() const;
312
313     //Work Functions
314     float constrain_angle(const float &angle); // puts angle from -pi to pi from stackoverflow.com
315     float angle_conv(const float &angle); // converts angle from -2pi to 2pi from stackoverflow.com
316     float angle_diff(const float &angle_a, const float &angle_b); // ensures a way to handle pi == -pi from stackoverflow.com
317     float unwrap_phase(const float &prev_angle, const float &current_angle); // unwrap function from stackoverflow.com
318     std::vector<float> calculate_fingerprints(const std::vector<gr_complex> &burst); // the input burst will return the est
        device
319
320     //make_subregion will return a subregion consisting of the amplitude, phse, and frequency of the provided samples
321     //likewise, the amplitude, phase, and frequency sub-sections will consist of 3 samples each of the calculated
322     //variance, skewness, and kurtosis
323     std::vector<float> make_subregion(const std::vector<gr_complex> &sub_region);
324
325     // fingerprint_emmitter will take in the calculated fingerprints statistics and
326     // will multiply the vector with the loaded fingerprint data
327     std::vector<float> fingerprint_emitter(const std::vector<float> &fingerprint);
328
329
330 };
331 } // namespace ieee802_15_4
332 } // namespace gr
333 #endif /* INCLUDED_IEEE802_15_4_DNA_DETECTOR_CCF_IMPL_H */
334
335 /*****
336  *dna_detector_ccf_imph.cc impletation source code
337  *****/
338 /* -- c++ -- */
339 /*
340  * Copyright 2018 gr-ieee802_15_4 author.
341  *
342  * This is free software; you can redistribute it and/or modify
343  * it under the terms of the GNU General Public License as published by
344  * the Free Software Foundation; either version 3, or (at your option)
345  * any later version.
346  *
347  * This software is distributed in the hope that it will be useful,
348  * but WITHOUT ANY WARRANTY; without even the implied warranty of
349  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
350  * GNU General Public License for more details.
351  *
352  * You should have received a copy of the GNU General Public License
353  * along with this software; see the file COPYING. If not, write to
354  * the Free Software Foundation, Inc., 51 Franklin Street,
355  * Boston, MA 02110-1301, USA.
356  */
357

```

```

358 #ifdef HAVE_CONFIG_H
359 #include "config.h"
360 #endif
361
362 #include <gnuradio/io_signature.h>
363 #include "dna_detector_ccf_impl.h"
364
365 namespace gr {
366     namespace ieee802_15_4 {
367         dna_detector_ccf::sptr
368         dna_detector_ccf::make(int devices, int dna_row_size, int dna_column_size, int dna_region_num,
369                               DebugType debug, float samp_rate, std::string fingerprint_filename,
370                               SWITCH fingerprint_save_switch, std::string class_matrix_filename,
371                               std::string means_matrix_filename, std::string norm_vector_filename,
372                               std::string xoffset_vector_filename, SWITCH class_matrix_load_switch,
373                               SWITCH save_stats_switch, SWITCH save_dna_results_switch)
374         {
375             return gnuradio::get_initial_sptr
376                 (new dna_detector_ccf_impl(devices, dna_row_size, dna_column_size, dna_region_num,
377                                           debug, samp_rate, fingerprint_filename,
378                                           fingerprint_save_switch, class_matrix_filename,
379                                           means_matrix_filename, norm_vector_filename,
380                                           xoffset_vector_filename, class_matrix_load_switch,
381                                           save_stats_switch, save_dna_results_switch));
382         }
383
384         /*
385          * The private constructor
386          */
387         dna_detector_ccf_impl::dna_detector_ccf_impl(int devices, int dna_row_size, int dna_column_size, int dna_region_num,
388                                                     DebugType debug, float samp_rate, std::string fingerprint_filename,
389                                                     SWITCH fingerprint_save_switch, std::string class_matrix_filename,
390                                                     std::string means_matrix_filename, std::string norm_vector_filename,
391                                                     std::string xoffset_vector_filename, SWITCH class_matrix_load_switch,
392                                                     SWITCH save_stats_switch, SWITCH save_dna_results_switch)
393             : gr::block("dna_detector_ccf",
394                         gr::io_signature::make(1, 1, sizeof(gr_complex)),
395                         gr::io_signature::make(0, 0, 0))
396         {
397             d_devices = devices;
398             d_dna_row_size = dna_row_size;
399             d_dna_column_size = dna_column_size;
400             d_dna_region_num = dna_region_num;
401             set_debug(debug);
402             d_samp_rate = samp_rate;
403             d_phy_802_15_4.set_sample_rate(samp_rate);
404             d_fingerprint_filename = fingerprint_filename;
405             d_fingerprint_save_switch = fingerprint_save_switch;
406             d_class_matrix_filename = class_matrix_filename;
407             d_class_matrix_load_switch = class_matrix_load_switch;
408             d_save_stats_switch = save_stats_switch;
409             d_save_dna_results_switch = save_dna_results_switch;
410             d_means_matrix_filename = means_matrix_filename;
411             d_norm_vector_filename = norm_vector_filename;

```

```

412     d_xoffset_vector_filename = xoffset_vector_filename;
413
414     // vector will be used to keep track of what device
415     // "bin's" were detected
416     for(int c = 0; c < devices; c++)
417     {
418         d_emitter_bin_counter.push_back(0);
419     }
420     std::cout << "\nSample Rate on DNA Detector set to " << samp_rate << std::endl;
421
422     // If the user doesn't load any dna data,
423     // the data will automatically be set
424     // as an Identity matrix
425     if(d_class_matrix_load_switch == SWITCH::ON)
426     {
427         // empty vectors will be used to store the loaded data
428         // the eigen map function will handle mapping the data
429         // as a vector/matrix as appropriate
430         std::vector<float> data_vector(dna_row_size * dna_column_size, 0.0f);
431         std::vector<float> means_vector(devices * dna_column_size, 0.0f);
432         std::vector<float> norm_vector(dna_row_size, 0.0f);
433         std::vector<float> offset_vector(dna_row_size, 0.0f);
434
435         d_dna_loader = std::make_shared<file_read_impl>(sizeof(float), class_matrix_filename.c_str(), false);
436         d_dna_loader->read_data(dna_row_size*dna_column_size, (void *)data_vector.data());
437         std::cout << "\n" << d_class_matrix_filename << " classification matrix loaded ";
438         std::cout << data_vector.size() << "-items.\n" << std::endl;
439         d_dna_matrix = EigMat<float>::Map(data_vector.data(),
440                                         dna_row_size,
441                                         dna_column_size);
442         std::cout << "\n" << dna_row_size << " by " << dna_column_size << " loaded.\n" << std::endl;
443
444         d_class_means_loader = std::make_shared<file_read_impl>(sizeof(float), means_matrix_filename.c_str(), false);
445         d_class_means_loader->read_data(devices*dna_column_size, (void *)means_vector.data());
446         std::cout << "\n" << d_means_matrix_filename << " means matrix loaded ";
447         std::cout << data_vector.size() << "-items.\n" << std::endl;
448         d_class_means_matrix = EigMat<float>::Map(means_vector.data(),
449                                                  devices,
450                                                  dna_column_size);
451         std::cout << "\n" << devices << " by " << dna_column_size << " loaded.\n" << std::endl;
452
453         d_norm_vector_loader = std::make_shared<file_read_impl>(sizeof(float), norm_vector_filename.c_str(), false);
454         d_norm_vector_loader->read_data(dna_row_size, (void *)norm_vector.data());
455         std::cout << "\n" << d_norm_vector_filename << " normalization vector loaded ";
456         std::cout << norm_vector.size() << "-items.\n" << std::endl;
457         d_norm_vector = EigRowVec<float>::Map(norm_vector.data(), 1, dna_row_size);
458         std::cout << "\n1 by " << dna_row_size << " loaded.\n" << std::endl;
459
460         d_xoffset_vector_loader = std::make_shared<file_read_impl>(sizeof(float), xoffset_vector_filename.c_str(), false);
461         d_xoffset_vector_loader->read_data(dna_row_size, (void *)offset_vector.data());
462         std::cout << "\n" << d_xoffset_vector_filename << " xoffset vector loaded ";
463         std::cout << offset_vector.size() << "-items.\n" << std::endl;
464         d_xoffset_vector = EigRowVec<float>::Map(offset_vector.data(), 1, dna_row_size);
465         std::cout << "\n1 by " << dna_row_size << " loaded.\n" << std::endl;

```

```

466
467 } else {
468     std::cout << "\nClassification data matrix wasn't loaded. A " << dna_row_size << " by " << dna_column_size << "
        identity matrix loaded instead." << "\n";
469     std::cout << "A pure " << dna_row_size << " by 1 fingerprint passthrough will occur\n" << std::endl;
470     d_dna_matrix = Eigen::MatrixXf::Identity(dna_row_size, dna_row_size);
471
472     std::cout << "\nMeans data matrix wasn't loaded. A " << devices << " by " << dna_column_size << " identity matrix
        loaded instead." << "\n";
473     std::cout << "A " << devices << " by " << dna_column_size << " identity matrix will be loaded instead.\n" << std::endl;
474     d_class_means_matrix = Eigen::MatrixXf::Identity( devices, dna_column_size);
475
476     std::cout << "\nNormalization data vector wasn't loaded. A 1 by " << dna_row_size << " ones vector loaded instead." <<
        "\n";
477     std::cout << "A 1 by " << dna_row_size << " ones vector will be loaded instead.\n" << std::endl;
478     d_norm_vector = Eigen::MatrixXf::Ones( 1, dna_row_size);
479
480     std::cout << "\nXOffset data vector wasn't loaded. A 1 by " << dna_row_size << " ones vector loaded instead." << "\n";
481     std::cout << "A 1 by " << dna_row_size << " ones vector will be loaded instead.\n" << std::endl;
482     d_xoffset_vector = Eigen::MatrixXf::Ones( 1, dna_row_size);
483
484 }
485
486 // The collected fingerprint data will continuously be
487 // stacked on the end of the file until the program is over
488 if(d_fingerprint_save_switch == SWITCH::ON)
489 {
490     std::cout << "\nGenerated fingerprints will be saved as binary data to " << d_fingerprint_filename << "\n" << std::endl;
491     d_dna_fingerprint_write = std::make_shared<file_store_impl>(sizeof(float), d_fingerprint_filename.c_str(),
492                                     //true, true);
493                                     true, false);
494 }
495
496 // The output dna analysis will be saved just like the
497 // fingerprints but with "_dna_results" appended to
498 // the default filename
499 if(d_save_dna_results_switch == SWITCH::ON)
500 {
501     std::string temp = fingerprint_filename + "_dna_results";
502     std::cout << "\nGenerated dna_results will be saved as binary data to " << temp << "\n" << std::endl;
503     d_dna_results_writer = std::make_shared<file_store_impl>(sizeof(float), temp.c_str(), true, false); //true, true);
504 }
505
506 // Option to collect the statistics generated
507 // for the generated fingerprints and results
508 if( d_save_stats_switch == SWITCH::ON)
509 {
510     // from stackoverflow.com, just a quick way to append time/date to a filename
511     auto t = std::time(nullptr);
512     auto tm = *std::localtime(&t);
513
514     // stat's result file load/creation
515     d_msg_stream << d_fingerprint_filename << "_stats_" << std::put_time(&tm, "%d%b%y_%R") << ".txt";
516     d_fingerprint_stats_filename.append(d_msg_stream.str());

```

```

517     clear_stream();
518     std::cout << "\nGenerated stats will be saved to " << d_fingerprint_stats_filename << ".\n" << std::endl;
519     d_stats_writer = std::make_shared<file_store_impl>(sizeof(char), d_fingerprint_stats_filename.c_str(),
520                                                     false, false);
521     d_msg_stream << "\n===== " << "\n";
522     d_msg_stream << "Collection started on " << std::put_time(&tm, "%F_%T");
523     d_msg_stream << "\nSample Rate at " << samp_rate;
524     d_msg_stream << "\n" << d_dna_region_num << " regions of interest for fingerprints = " << dna_region_num * 9 << "
        samples per fingerprint";
525     d_msg_stream << "\nPreamble sample length at " << samp_rate << " Samp/sec is " << d_phy_802_15_4.
        get_preamble_len_samples();
526     d_msg_stream << "\nPacket sample length at " << samp_rate << " Samp/sec is " << d_phy_802_15_4.get_packet_len_samples()
        ;
527     d_msg_stream << "\n===== " << "\n";
528     d_stats_writer->store_data(d_msg_stream.str().size(), (void*)d_msg_stream.str().c_str());
529     clear_stream();
530
531
532     // matrix result file load/creation
533     d_msg_stream << d_fingerprint_filename << "_matrix_mult_" << std::put_time(&tm, "%F_%T") << ".txt";
534     d_matrix_mult_results_filename.append(d_msg_stream.str());
535     clear_stream();
536     d_matrix_results_writer = std::make_shared<file_store_impl>(sizeof(char), d_matrix_mult_results_filename.c_str(),
537                                                                 false, false);
538     std::cout << "\nGenerated multiplication results will be saved to " << d_matrix_mult_results_filename << ".\n" << std::
        endl;
539
540
541     d_msg_stream << "\n===== " << "\n";
542     d_msg_stream << "Collection started on " << std::put_time(&tm, "%F_%T");
543     d_msg_stream << "\nSample Rate at " << samp_rate << "\n";
544     d_msg_stream << "\nPreamble sample length at " << samp_rate << " Samp/sec is " << d_phy_802_15_4.
        get_preamble_len_samples();
545     d_msg_stream << "\nPacket sample length at " << samp_rate << " Samp/sec is " << d_phy_802_15_4.get_packet_len_samples()
        ;
546     d_msg_stream << "\n" << d_dna_region_num << " regions of interest for fingerprints = " << dna_region_num * 9 << "
        samples per fingerprint";
547     d_msg_stream << "\nLoaded Classification Matrix Dimensions: " << d_dna_matrix.rows() << " by " << d_dna_matrix.cols() <<
        "\n";
548     d_msg_stream << d_dna_matrix << "\n";
549     d_msg_stream << "\n===== " << "\n";
550     d_msg_stream << "Loaded Means Matrix Dimensions: " << d_class_means_matrix.rows() << " by " << d_class_means_matrix.cols
        () << "\n";
551     d_msg_stream << d_class_means_matrix << "\n";
552     d_msg_stream << "\n===== " << "\n";
553     d_msg_stream << "Loaded Normalization Vector Dimensions: 1 by " << d_norm_vector.cols() << "\n";
554     d_msg_stream << d_norm_vector << "\n";
555     d_msg_stream << "\n===== " << "\n";
556     d_msg_stream << "Loaded X-Offset Vector Dimensions: 1 by " << d_xoffset_vector.cols() << "\n";
557     d_msg_stream << d_xoffset_vector << "\n";
558     d_msg_stream << "\n===== " << "\n";
559     d_matrix_results_writer->store_data(d_msg_stream.str().size(), (void*)d_msg_stream.str().c_str());
560 }
561

```

```

562     // Option to save DNA results
563
564     burst_search();
565     message_port_register_out(pmt::mp("dna_out"));
566 }
567
568 /*
569  * Our virtual destructor.
570  */
571 dna_detector_ccf_impl::~dna_detector_ccf_impl()
572 {
573 }
574
575 void
576 dna_detector_ccf_impl::forecast (int noutput_items, gr_vector_int &ninput_items_required)
577 {
578     ninput_items_required[0] = 2044;
579 }
580
581 int
582 dna_detector_ccf_impl::general_work (int noutput_items,
583                                     gr_vector_int &ninput_items,
584                                     gr_vector_const_void_star &input_items,
585                                     gr_vector_void_star &output_items)
586 {
587     const gr_complex *input_bursts = (const gr_complex *) input_items[0]; // detected ieee802-15-4 bursts
588     int ninput = ninput_items[0]; // size of input burst data stream
589
590     float magnitude = 0.0f;
591
592     int burst_index = 0;
593     int sample_limit = d_phy_802_15_4.get_preamble_len_samples() + 10; // "oh crap" 10 sample buffer added
594
595     while(burst_index < ninput)
596     {
597         switch(d_state)
598         {
599             case SEARCHING:
600                 //since the bursts are zero buffered and dealing with floating numbers,
601                 //an extremely small floating number will be used
602                 if(std::abs(input_bursts[burst_index]) > 1.0e-38f)
603                 {
604                     // The previous detector ensures that every burst has exactly a complex
605                     // zero as thier value so all complex zero's are ignored before the
606                     // preamble bursts
607                     burst_detect();
608                 } else {
609                     burst_index++;
610                 }
611                 break;
612
613             case DETECTED:
614                 //if(d_current_detected_preamble.get_sample_length() < sample_limit)
615                 if(d_current_detected_preamble.size() < sample_limit)

```

```

616     {
617         d_current_detected_preamble.push_back(input_bursts[burst_index++]);
618     }else{
619         // filled burst will be sent to be fingerprinted
620         burst_fingerprint();
621     }
622     break;
623
624 case FINGERPRINT:
625     //put case statement in brackets because of
626     //compilation errors
627     {
628         // At this point, I need to perform RFDNA Analysis and get the fingerprint
629         // from the latest pre-amble
630         d_dna_fingerprints = calculate_fingerprints(d_current_detected_preamble);
631
632         // Here, the fingerprint will be compared against the trained
633         // data and return the euclidean distances calculated from
634         // the loaded Matlab matrices
635         d_dna_results = fingerprint_emitter(d_dna_fingerprints);
636
637         //We need to check if a NaN value is detected since it would invalidate our fingerprint
638         bool nan_found = false;
639         for(std::vector<gr_complex>::size_type i = 0; i != d_dna_fingerprints.size(); i++)
640         {
641             if(std::isnan(d_dna_fingerprints[i]))
642             {
643                 nan_found = true;
644                 d_nan_counter++;
645                 break;
646             }
647         }
648
649         //Increment counters for stats
650         if(!nan_found)
651         {
652             d_fingerprints_counter++;
653         }
654
655         // Here, the generated fingerprint will be passed on if
656         // the user has enabled saving fingerprints and they do
657         // not contain NaN's
658         if((d_fingerprint_save_switch == SWITCH::ON) && !nan_found)
659         {
660             d_dna_fingerprint_write->store_data(d_dna_fingerprints.size(), (void*)d_dna_fingerprints.data());
661         }
662
663         // Here, the generated fingerprint will be passed on if
664         // the user has enabled saving fingerprints and they do
665         // not contain NaN's
666         if((d_save_dna_results_switch == SWITCH::ON) && !nan_found)
667         {
668             d_dna_results_writer->store_data(d_dna_results.size(), (void*)d_dna_results.data());
669         }

```

```

670
671 // setting the initial min to the numeric limit
672 // of a float ensures that
673 float min = std::numeric_limits<float>::max();
674 int most_likely = 0;
675
676 if (!nan_found)
677 {
678     for(int c = 0; c < d_devices; c++)
679     {
680         if (d_dna_results[c] < min)
681         {
682             min = d_dna_results[c];
683             most_likely = c+1;
684         }
685     }
686 }
687
688 //Increment Emitter array
689 d_emitter_bin_counter[most_likely - 1] += 1;
690
691 //now I use c to pass the vector values to the histogram
692 pmt::pmt_t dna_results = pmt::make_f32vector(d_devices,0.0);
693 for(int i = 0; i < d_devices; i++)
694 {
695     pmt::f32vector_set(dna_results, i, most_likely);
696 }
697
698 // A message indicating which device will only be sent out
699 // if the fingerprint generated was valid
700 if (!nan_found)
701 {
702     message_port_pub(pmt::mp("dna_out"), dna_results);
703 }
704
705 // This if block will save the data overview of the fingerprints
706 // in a simple to read output
707 if (d_save_stats_switch == SWITCH::ON)
708 {
709     clear_stream();
710     d_msg_stream << "\n-----" <<
711         "\n";
712     d_msg_stream << "Burst: " << d_burst_count + 1 << "\n";
713     d_msg_stream << "Device Most Likely: " << (nan_found ? "N/A, NaN detected" : std::to_string(most_likely)) << "\n";
714     d_msg_stream << "Device Result Value: " << (nan_found ? "NaN" : std::to_string(d_dna_results[most_likely - 1]))
715         << "\n";
716     d_msg_stream << "Total Valid Fingerprints: " << d_fingerprints_counter << "\n";
717     d_msg_stream << "Total Invalid Fingerprints: " << d_nan_counter << "\n";
718     for(std::vector<unsigned long int>::size_type i = 0; i != d_emitter_bin_counter.size(); i++)
719     {
720         d_msg_stream << "Emitter_" << i+1 << " Count:" << d_emitter_bin_counter[i] << "\t";
721     }

```

```

720     d_msg_stream << "\n-----" <<
721         "\n";
722     d_stats_writer->store_data(d_msg_stream.str().size(),(void*)d_msg_stream.str().c_str());
723 }
724
725 // this if statement will keep track of all the math involved and results for every
726 // fingerprint that is analyzed
727 if(d_save_stats_switch == SWITCH::ON)
728 {
729     clear_stream();
730     d_msg_stream << "\n-----" <<
731         "\n";
732     d_msg_stream << "Burst: " << d_burst_count + 1 << (nan_found ? " contains a NaN\n" : "\n");
733     d_msg_stream << "Fingerprints: ";
734     for(std::vector<float>::size_type i = 0; i != d_dna_fingerprints.size(); i++)
735     {
736         if(i < (d_dna_fingerprints.size() - 1))
737         {
738             d_msg_stream << d_dna_fingerprints[i] << ", ";
739         }else{
740             d_msg_stream << d_dna_fingerprints[i] << "\n";
741         }
742     }
743     d_msg_stream << "DNA Results: ";
744     for(std::vector<float>::size_type i = 0; i != d_dna_results.size(); i++)
745     {
746         if(i < (d_dna_results.size() - 1))
747         {
748             d_msg_stream << d_dna_results[i] << ", ";
749         }else{
750             d_msg_stream << d_dna_results[i] << "\n";
751         }
752     }
753     d_msg_stream << "-----" << "\n";
754
755     d_matrix_results_writer->store_data(d_msg_stream.str().size(),(void*)d_msg_stream.str().c_str());
756 }
757
758 //Option to actively see the fingerprints & results comming out
759 if((d_debug == DebugType::USRP_ON) || (d_debug == DebugType::ALL_ON))
760 {
761     std::cout << "\n-----" << "\n";
762
763     std::cout << "Fingerprint #" << d_burst_count + 1 << ": ";
764     for(std::vector<float>::iterator c = d_dna_fingerprints.begin(); c != d_dna_fingerprints.end(); ++c)
765     {
766         std::cout << (*c) << ", ";
767     }
768     std::cout << "\nResults #" << d_burst_count + 1 << ": ";
769     for(std::vector<float>::iterator c = d_dna_results.begin(); c != d_dna_results.end(); ++c)
770     {
771         std::cout << (*c) << ", ";
772     }
773 }

```

```

769         std::cout << "\n-----" << std
              :: endl;
770     }
771
772     d_burst_count++; // Keeping track for statistics
773
774     burst_search();
775     break;
776 }
777 default:
778     assert(0);
779     break;
780
781 }
782 }
783
784 if ((d_debug == DebugType::PACKET_USRP_ON) || (d_debug == DebugType::ALL_ON))
785 {
786     std::cout << "\n";
787     std::cout << "Current Preamble Sample Len: " << d_phy_802_15_4.get_preamble_len_samples() << "\n";
788     std::cout << "Current Input Size: " << ninput << "\n";
789     std::cout << "Current Burst Count: " << d_burst_count << "\n";
790     std::cout << "Current Valid Fingerprints: " << d_fingerprints_counter << "\n";
791     std::cout << "Current Invalid Fingerprints: " << d_nan_counter << "\n";
792     std::cout << "\n-----" << std::endl;
793 }
794
795 consume(0, noutput_items);
796
797 // Tell runtime system how many output items we produced.
798 return 0;
799 }
800
801 void
802 dna_detector_ccf_impl::burst_search()
803 {
804     d_state = SEARCHING;
805     d_current_detected_preamble.clear();
806 }
807
808 void
809 dna_detector_ccf_impl::burst_detect()
810 {
811     d_state = DETECTED;
812 }
813
814 void
815 dna_detector_ccf_impl::burst_fingerprint()
816 {
817     // At this point, I need to perform RFDNA Analysis
818     //this is to get rid of the extra zeroes beyond the preamble sample size
819     if (d_current_detected_preamble.size() > d_phy_802_15_4.get_preamble_len_samples())
820     {
821         d_current_detected_preamble.resize(d_phy_802_15_4.get_preamble_len_samples());

```

```

822     }
823     d_state = FINGERPRINT;
824 }
825
826 void
827 dna_detector_ccf_impl::set_devices(const int &devices){
828     d_devices = devices;
829 }
830
831 void
832 dna_detector_ccf_impl::set_dna_row_size(const int &dna_row_size)
833 {
834     d_dna_row_size = dna_row_size;
835 }
836
837 void
838 dna_detector_ccf_impl::set_dna_column_size(const int &dna_column_size)
839 {
840     d_dna_column_size = dna_column_size;
841 }
842
843 void
844 dna_detector_ccf_impl::set_dna_region_num(const int &dna_region_num)
845 {
846     d_dna_region_num = dna_region_num;
847 }
848
849 void
850 dna_detector_ccf_impl::set_debug(const DebugType &debug)
851 {
852     d_debug = debug;
853     d_usrp_debug = (debug & USRP_ON) > 0 ? 1 : 0;
854 }
855
856 void
857 dna_detector_ccf_impl::set_dna_fingerprint_filename(const std::string &fingerprint_filename)
858 {
859     d_fingerprint_filename = fingerprint_filename;
860 }
861
862 void
863 dna_detector_ccf_impl::set_dna_fingerprint_save_switch(const SWITCH &fingerprint_save_switch)
864 {
865     d_fingerprint_save_switch = fingerprint_save_switch;
866 }
867
868 void
869 dna_detector_ccf_impl::set_dna_classification_matrix_filename(const std::string &class_matrix_filename)
870 {
871     d_class_matrix_filename = class_matrix_filename;
872 }
873
874 void
875 dna_detector_ccf_impl::set_dna_means_matrix_filename(const std::string &means_matrix_filename)

```

```

876     {
877         d_means_matrix_filename = means_matrix_filename;
878     }
879
880     void
881     dna_detector_ccf_impl::set_dna_classification_load_switch(const SWITCH &class_matrix_load_switch)
882     {
883         d_class_matrix_load_switch = class_matrix_load_switch;
884     }
885
886     void
887     dna_detector_ccf_impl::set_save_stats_switch(const SWITCH &save_stats_switch)
888     {
889         d_save_stats_switch = save_stats_switch;
890     }
891
892     void
893     dna_detector_ccf_impl::set_save_dna_results_switch(const SWITCH &save_dna_results_switch)
894     {
895         d_save_dna_results_switch = save_dna_results_switch;
896     }
897
898     void
899     dna_detector_ccf_impl::set_dna_norm_vector_filename(const std::string &norm_vector_filename)
900     {
901         d_norm_vector_filename = norm_vector_filename;
902     }
903
904     void
905     dna_detector_ccf_impl::set_dna_xoffset_vector_filename(const std::string &xoffset_vector_filename)
906     {
907         d_xoffset_vector_filename = xoffset_vector_filename;
908     }
909
910     void
911     dna_detector_ccf_impl::set_samp_rate(const float &samp_rate)
912     {
913         d_samp_rate = samp_rate;
914     }
915
916     int
917     dna_detector_ccf_impl::get_devices() const
918     {
919         return d_devices;
920     }
921
922     int
923     dna_detector_ccf_impl::get_dna_row_size() const
924     {
925         return d_dna_row_size;
926     }
927
928     int
929     dna_detector_ccf_impl::get_dna_column_size() const

```

```

930     {
931         return d_dna_column_size;
932     }
933
934     int
935     dna_detector_ccf_impl::get_dna_region_num() const
936     {
937         return d_dna_region_num;
938     }
939
940     DebugType
941     dna_detector_ccf_impl::get_debug() const
942     {
943         return d_debug;
944     }
945
946     std::string
947     dna_detector_ccf_impl::get_dna_fingerprint_filename() const
948     {
949         return d_fingerprint_filename;
950     }
951
952     SWITCH
953     dna_detector_ccf_impl::get_dna_fingerprint_save_switch() const
954     {
955         return d_fingerprint_save_switch;
956     }
957
958     std::string
959     dna_detector_ccf_impl::get_dna_classification_matrix_filename() const
960     {
961         return d_class_matrix_filename;
962     }
963
964     std::string
965     dna_detector_ccf_impl::get_dna_means_matrix_filename() const
966     {
967         return d_means_matrix_filename;
968     }
969
970     SWITCH
971     dna_detector_ccf_impl::get_dna_classification_load_switch() const
972     {
973         return d_class_matrix_load_switch;
974     }
975
976     SWITCH
977     dna_detector_ccf_impl::get_save_stats_switch() const
978     {
979         return d_save_stats_switch;
980     }
981
982     SWITCH
983     dna_detector_ccf_impl::get_save_dna_results_switch() const

```

```

984     {
985         return d_save_dna_results_switch;
986     }
987
988     float
989     dna_detector_ccf_impl::get_samp_rate() const
990     {
991         return d_samp_rate;
992     }
993
994     std::string
995     dna_detector_ccf_impl::get_dna_norm_vector_filename() const
996     {
997         return d_norm_vector_filename;
998     }
999
1000    std::string
1001    dna_detector_ccf_impl::get_dna_xoffset_vector_filename() const
1002    {
1003        return d_xoffset_vector_filename;
1004    }
1005
1006    float
1007    dna_detector_ccf_impl::constrain_angle(const float &angle)
1008    {
1009        float x = fmod(angle + M_PI, 2*M_PI);
1010        if (x < 0)
1011        {
1012            x += 2*M_PI;
1013        }
1014
1015        return x - M_PI;
1016    }
1017
1018    float
1019    dna_detector_ccf_impl::angle_conv(const float &angle)
1020    {
1021        return fmod(constrain_angle(angle), 2*M_PI);
1022    }
1023
1024    float
1025    dna_detector_ccf_impl::angle_diff(const float &angle_a, const float &angle_b)
1026    {
1027        float dif = fmod(angle_b - angle_a + M_PI, 2*M_PI);
1028
1029        if (dif < 0)
1030        {
1031            dif += (2*M_PI);
1032        }
1033        return std::move(dif - M_PI);
1034    }
1035
1036    float
1037    dna_detector_ccf_impl::unwrap_phase(const float &prev_angle, const float &current_angle)

```

```

1038     {
1039         return std::move(prev_angle - angle_diff(current_angle, angle_conv(prev_angle)));
1040     }
1041
1042     std::vector<float>
1043     dna_detector_ccf_impl::calculate_fingerprints(const std::vector<gr_complex> &burst)
1044     {
1045         // Assuming that the sampling rate is @ 4MS/s, then there should be 640 samples.
1046         // That preamble will be broken down into 64-samples analyzing the variance, phase,
1047         // and frequencies variance, skew, and kurtosis such that 9-samples will be produced for
1048         // every 64-samples.
1049         //
1050         //           Amplitude           Phase(rads)           Frequency(rads/sec)
1051         // [Variance, Skewness, Kurtosis] [Variance, Skewness, Kurtosis] [Variance, Skewness, Kurtosis]
1052         //
1053         // For example, if each returning block of 9-samples represents 1-subregion and 10-subregion represents a fingerprint
1054         // i.e. 90-samples returned @ 4MS/sec
1055
1056         std::vector<float> fingerprint;
1057         int sub_samples = d_phy_802_15_4.get_preamble_len_samples()/d_dna_region_num;
1058
1059         std::vector<gr_complex>::const_iterator i;
1060         std::vector<float> subregion_temp;
1061         for(int c = 0; c < d_dna_region_num; c++)
1062         {
1063             i = burst.begin(); //setting the iterator to the beginning of the burst
1064
1065             // Next, the iterator will be offset by a multiple of the subregion size
1066             // based on the sample rate and passed into an rvalue vector to make
1067             // the subregion
1068             subregion_temp = std::move(make_subregion(
1069                 std::vector<gr_complex>(i + sub_samples*c,
1070                     i + (c + 1)*sub_samples)));
1071
1072             // Finally, we append the sub_region to the fingerprint vector
1073             fingerprint.insert(fingerprint.end(), subregion_temp.begin(), subregion_temp.end());
1074         }
1075
1076         return std::move(fingerprint);
1077     }
1078
1079     // Accumulators and burst floats were cast to doubles for additional precision
1080     std::vector<float>
1081     dna_detector_ccf_impl::make_subregion(const std::vector<gr_complex> &burst)
1082     {
1083         std::vector<float> sub_region;
1084         std::vector<double> temp_phase;
1085
1086         // complex data was converted to doubles for statistical calculations
1087         // for additional precision
1088         std::vector<dna_marker<double>> stats;
1089         for(int c = 0; c < 3; c++)
1090         {
1091             stats.push_back(std::move(dna_marker<double>()));

```

```

1092     }
1093
1094     // prev_phase and unwrapped phase will be used to keep
1095     // track of how many times the IQ unit circle has been
1096     // traversed
1097     double prev_phase = 0.0;
1098     double unwrapped_phase = 0.0;
1099     for(std::vector<gr_complex>::size_type i = 0; i != burst.size(); i++)
1100     {
1101         unwrapped_phase = unwrap_phase(prev_phase, std::arg(burst[i]));
1102         stats[0](std::abs(burst[i]));
1103         stats[1](unwrapped_phase);
1104         stats[2](d_samp_rate * (unwrapped_phase - prev_phase));
1105         prev_phase = unwrapped_phase;
1106     }
1107
1108     // After all the processing is done, 9-samples (3 per feature) will be pushed
1109     // onto the vector and returned. Each output was cast back to floats after
1110     // all of the statistics were tabulated
1111     for(int c = 0; c < stats.size(); c++)
1112     {
1113         sub_region.push_back(variance(stats[c]));
1114         sub_region.push_back(skewness(stats[c]));
1115         sub_region.push_back(kurtosis(stats[c]));
1116     }
1117
1118     return std::move(sub_region);
1119 }
1120
1121 std::vector<float>
1122 dna_detector_ccf_impl::fingerprint_emitter(const std::vector<float> &fingerprint)
1123 {
1124     std::vector<float> emitter_results;
1125
1126     auto fingerprint_vector = EigRowVec<float>::Map(fingerprint.data(), fingerprint.size());
1127     auto adjusted_fingerprint_vector = fingerprint_vector - d_xoffset_vector;
1128     auto results = (adjusted_fingerprint_vector.cwiseProduct(d_norm_vector))* d_dna_matrix;
1129
1130     for(int c = 0; c < d_devices; c++)
1131     {
1132         // using Euclidean distance,  $c = (a^2 - b^2)^{.5}$ ,
1133         // by pulling a row of the means matrix that corresponds to each device,
1134         // perform an element-wise difference and squaring of the differences,
1135         // summing the squared difference results, and finally getting the root
1136         // of the sum.
1137         auto row_diff = d_class_means_matrix.row(c) - results;
1138         auto row_squared = row_diff.cwiseProduct(row_diff);
1139         auto row_sum = row_squared.rowwise().sum();
1140         auto row_sqrt = row_sum.cwiseSqrt();
1141         auto row_results = row_sqrt.value();
1142
1143         emitter_results.push_back(row_results);
1144     }
1145

```

```

1146         return std::move(emitter_results);
1147     }
1148
1149     void
1150     dna_detector_ccf_impl::clear_stream()
1151     {
1152         d_msg_stream.str(std::string());
1153     }
1154 } /* namespace ieee802_15_4 */
1155 } /* namespace gr */

```

D.2.2 Dna detector ccf XML GRC Code.

```

1  <?xml version="1.0"?>
2  <block>
3      <name>Dna detector ccf</name>
4      <key>ieee802_15_4_dna_detector_ccf</key>
5      <category>[IEEE802_15_4]</category>
6      <import>import ieee802_15_4</import>
7      <make>ieee802_15_4.dna_detector_ccf($devices, $dna_row_size, $dna_col_size, $dna_region_num,
8                                          $debug, $samp_rate, $fingerprint_filename,
9                                          $fingerprint_save_switch, $class_matrix_filename,
10                                         $means_matrix_filename, $norm_vector_filename,
11                                         $xoffset_vector_filename, $class_matrix_load_switch,
12                                         $save_stats_switch, $dna_save_switch)</make>
13      <param>
14          <name>Num of Devices</name>
15          <key>devices</key>
16          <type>int</type>
17      </param>
18      <param>
19          <name>DNA File Row Size</name>
20          <key>dna_row_size</key>
21          <type>int</type>
22      </param>
23      <param>
24          <name>DNA File Col Size</name>
25          <key>dna_col_size</key>
26          <type>int</type>
27      </param>
28      <param>
29          <name>Number of Regions</name>
30          <key>dna_region_num</key>
31          <type>int</type>
32      </param>
33      <param>
34          <name>Debug</name>
35          <key>debug</key>
36          <type>enum</type>
37          <option>
38              <name>ALL_OFF</name>
39              <key>0</key>

```

```
40     </option>
41     <option>
42         <name>CHIP_ON</name>
43         <key>1</key>
44     </option>
45     <option>
46         <name>PACKET_ON</name>
47         <key>2</key>
48     </option>
49     <option>
50         <name>USRP_ON</name>
51         <key>4</key>
52     </option>
53     <option>
54         <name>CHIP_PACKET_ON</name>
55         <key>3</key>
56     </option>
57     <option>
58         <name>CHIP_USRP_ON</name>
59         <key>5</key>
60     </option>
61     <option>
62         <name>PACKET_USRP_ON</name>
63         <key>6</key>
64     </option>
65     <option>
66         <name>ALL_ON</name>
67         <key>7</key>
68     </option>
69 </param>
70 <param>
71     <name>Samp_rate</name>
72     <key>samp_rate</key>
73     <type>float </type>
74 </param>
75 <param>
76     <name>Default Fingerprint Filename</name>
77     <key>fingerprint_filename</key>
78     <type>file_save </type>
79 </param>
80 <param>
81     <name>Save Generated Fingerprints?</name>
82     <key>fingerprint_save_switch</key>
83     <type>enum</type>
84     <option>
85         <name>OFF</name>
86         <key>0</key>
87     </option>
88     <option>
89         <name>ON</name>
90         <key>1</key>
91     </option>
92 </param>
93 <param>
```

```
94     <name>Save DNA Results?</name>
95     <key>dna_save_switch </key>
96     <type>enum</type>
97     <option>
98         <name>OFF</name>
99         <key>0</key>
100     </option>
101     <option>
102         <name>ON</name>
103         <key>1</key>
104     </option>
105 </param>
106 <param>
107     <name>Default Classification Matrix</name>
108     <key>class_matrix_filename </key>
109     <type>file_open </type>
110 </param>
111 <param>
112     <name>Default Means Matrix</name>
113     <key>means_matrix_filename </key>
114     <type>file_open </type>
115 </param>
116 <param>
117     <name>Default Norm Vector</name>
118     <key>norm_vector_filename </key>
119     <type>file_open </type>
120 </param>
121 <param>
122     <name>Default XOffset Vector</name>
123     <key>xoffset_vector_filename </key>
124     <type>file_open </type>
125 </param>
126
127
128 <param>
129     <name>Load Classification Matrix?</name>
130     <key>class_matrix_load_switch </key>
131     <type>enum</type>
132     <option>
133         <name>OFF</name>
134         <key>0</key>
135     </option>
136     <option>
137         <name>ON</name>
138         <key>1</key>
139     </option>
140 </param>
141 <param>
142     <name>Save Generated Statistics?</name>
143     <key>save_stats_switch </key>
144     <type>enum</type>
145     <option>
146         <name>OFF</name>
147         <key>0</key>
```

```

148     </option>
149     <option>
150         <name>ON</name>
151         <key>1</key>
152     </option>
153 </param>
154 <sink>
155     <name>in</name>
156     <type>complex</type>
157 </sink>
158 <source>
159     <name>dna_out</name>
160     <type>message</type>
161 </source>
162 </block>

```

D.3 DNA Buffer Class

```

1  /*****
2      ***
3      *dna_buffer Class Definition
4      *****/
5  /*!
6   *   Class: dna_buffer
7   *   Author: Cruz, Frankie
8   *   Description: Extended vector template to buffer input data with markers
9   *                   to give a user a beginning/end of a buffer and keep track
10  *                   of dna preambles
11  *   Generated: Tues Nov 20 12:10:18 2018
12  *   Modified: Dec 13 12:00:37 2018
13  *   Build Version: 1.5
14  *   Changes: -buffer will now keep an array of sample lengths to pop off
15  *               the samples once a buffer threshold has been exceeded
16  *               -fixed input error with erase_buffer_element
17  */
18  class dna_buffer
19  {
20  public:
21      /*!
22       *   constructors and destructors
23       */
24      dna_buffer();
25      dna_buffer(const int &space); // constructor w/ blank space intervals
26      ~dna_buffer();
27
28      /*!
29       *   copy constructors and assignment operators
30       */
31      dna_buffer(const dna_buffer &copy);
32      dna_buffer &operator=(const dna_buffer &copy);

```

```

32     dna_buffer &operator=(const std::vector<gr_complex> &copy);
33
34     /*!
35     * move constructors and assignment operators
36     */
37     dna_buffer(dna_buffer && moved);
38     dna_buffer &operator=(dna_buffer &&moved);
39     dna_buffer &operator=(std::vector<gr_complex> &&moved);
40
41
42     /*!
43     * setters
44     */
45     void set_buffer();
46     void set_spacing(const int &space); // sets spacing size
47     void unset_buffer(); // deactivates the dna_buffer and clears out values
48     void erase_buffer_element(const int &element); // wrapper for vector erase function
49     void erase_buffer_range(const int &front, const int &back); // wrapper for vector erase function
50     void insert_rear_spacing(const int &space); // inserts a given zero buffer at the end of the buffer
51     void insert_front_spacing(const int &space); // insert a given zero buffer at the front of the buffer
52     void append_burst_sample_size(const int &size); // insert the size of the sample stored
53     void pop_front_burst_samples(); // remove # of samples from the buffer and remove the element the sample size
54
55     /*!
56     * getters
57     */
58     bool is_activated() const; // returns true if the buffer is set
59     int get_space() const; // returns the spacing default
60     int get_sample_length() const; // returns how long the buffer is in samples
61     int get_burst_length() const; // returns how many bursts are stored in the buffer
62     gr_complex operator[](const int &index) const; //get access to individual data in buffer
63     std::vector<gr_complex> get_stored_buffer() const; //returns a copy of the stored buffer
64
65     /*!
66     * append_preamble_points will be used in the case where more data has to be
67     * appended i.e. input buffer is large enough to collect more than one
68     * transmission
69     */
70     void append_buffer_data(const dna_buffer &append);
71     void append_buffer_data(const std::vector<gr_complex> &append);
72     void append_buffer_data(const gr_complex &element);
73
74     /*!
75     * store_preamble will take the input size and pointer to
76     * a vector and store it into the buffer space
77     */
78     void store_preamble_data(const int &ninput_size,
79                             const gr_complex *in_rx);
80
81     protected:
82     std::vector<gr_complex> d_buffer;
83     std::vector<int> d_sample_size_array; // keeps track of front bursts sample size
84     bool d_dna_buffer_on; // keeps track of buffer if it's on/off
85     int d_space; //default preamble spacing

```

```

86  };
87
88  /*****
      ***
89  *dna_buffer Source Code
90  *****/
      **/
91  dna_buffer::dna_buffer()
92  {
93      d_dna_buffer_on = false;
94      d_space = 0;
95  }
96
97  dna_buffer::dna_buffer(const int &space)
98  {
99      d_dna_buffer_on = false;
100     set_spacing(space);
101 }
102
103 dna_buffer::~dna_buffer()
104 {
105
106 }
107
108 dna_buffer::dna_buffer(const dna_buffer &copy)
109 {
110     d_dna_buffer_on = copy.d_dna_buffer_on;
111     d_space = copy.d_space;
112     d_buffer = copy.d_buffer;
113     d_sample_size_array = copy.d_sample_size_array;
114 }
115
116 dna_buffer
117 &dna_buffer::operator=(const dna_buffer &copy)
118 {
119     if(&copy != this)
120     {
121         d_dna_buffer_on = copy.d_dna_buffer_on;
122         d_space = copy.d_space;
123         d_buffer = copy.d_buffer;
124         d_sample_size_array = copy.d_sample_size_array;
125     }
126
127     return *this;
128 }
129
130 dna_buffer
131 &dna_buffer::operator=(const std::vector<gr_complex> &copy)
132 {
133     if(copy != d_buffer)
134     {
135         d_dna_buffer_on = true;
136         d_buffer = copy;
137     }

```

```

138 }
139
140 dna_buffer::dna_buffer(dna_buffer &&moved)
141 {
142     d_dna_buffer_on = std::move(moved.d_dna_buffer_on);
143     d_space = std::move(moved.d_space);
144     d_buffer = std::move(moved.d_buffer);
145     d_sample_size_array = std::move(moved.d_sample_size_array);
146 }
147
148 dna_buffer
149 &dna_buffer::operator=(dna_buffer &&moved)
150 {
151     if (this != &moved)
152     {
153         d_dna_buffer_on = std::move(moved.d_dna_buffer_on);
154         d_space = std::move(moved.d_space);
155         d_buffer = std::move(moved.d_buffer);
156         d_sample_size_array = std::move(moved.d_sample_size_array);
157     }
158
159     return *this;
160 }
161
162 dna_buffer
163 &dna_buffer::operator=(std::vector<gr_complex> &&moved)
164 {
165     set_buffer();
166     d_buffer = std::move(moved);
167 }
168
169 void
170 dna_buffer::set_buffer()
171 {
172     d_dna_buffer_on = true;
173 }
174
175 void
176 dna_buffer::set_spacing(const int &space)
177 {
178     d_space = space < 0 ? 0 : space;
179 }
180
181 void
182 dna_buffer::unset_buffer()
183 {
184     d_dna_buffer_on = false;
185     d_buffer.clear();
186     d_sample_size_array.clear();
187 }
188
189 void
190 dna_buffer::erase_buffer_element(const int &element)
191 {

```

```

192     d_buffer.erase(d_buffer.begin() + element);
193 }
194
195 void
196 dna_buffer::erase_buffer_range(const int &front, const int &back)
197 {
198     d_buffer.erase(d_buffer.begin() + front,
199                   d_buffer.begin() + back);
200 }
201
202 void
203 dna_buffer::insert_rear_spacing(const int &space)
204 {
205     if (space > 0)
206     {
207         for(int c = 0; c < space; c++)
208         {
209             d_buffer.push_back(gr_complex(0.0f, 0.0f));
210         }
211     }
212 }
213
214 void
215 dna_buffer::insert_front_spacing(const int &space)
216 {
217     if (space > 0)
218     {
219         for(int c = 0; c < space; c++)
220         {
221             d_buffer.insert(d_buffer.begin(), gr_complex(0.0f, 0.0f));
222         }
223     }
224 }
225
226 void
227 dna_buffer::append_burst_sample_size(const int &size)
228 {
229     d_sample_size_array.push_back(size);
230 }
231
232 void
233 dna_buffer::pop_front_burst_samples()
234 {
235     // have to have a check to see if there's anything
236     // in the buffer or we could possibly crash everything
237     // trying to remove a non-existent array
238     if (!d_sample_size_array.empty())
239     {
240         // pop off front # samples from oldest burst in the buffer
241         d_buffer.erase(d_buffer.begin(), d_buffer.begin() + d_sample_size_array[0]);
242
243         // pop off front of container holding # of samples of previous burst
244         d_sample_size_array.erase(d_sample_size_array.begin());
245     }

```

```

246 }
247
248 bool
249 dna_buffer::is_activated() const
250 {
251     return d_dna_buffer_on;
252 }
253
254 int
255 dna_buffer::get_space() const
256 {
257     return d_space;
258 }
259
260 int
261 dna_buffer::get_sample_length() const
262 {
263     return d_buffer.size();
264 }
265
266 int
267 dna_buffer::get_burst_length() const
268 {
269     return d_sample_size_array.size();
270 }
271
272 gr_complex
273 dna_buffer::operator[](const int &index) const
274 {
275     return d_buffer[index];
276 }
277
278 std::vector<gr_complex>
279 dna_buffer::get_stored_buffer() const
280 {
281     //if the vector is empty
282     return (d_buffer.empty()) ? std::vector<gr_complex>() : d_buffer;
283 }
284
285 void
286 dna_buffer::append_buffer_data(const dna_buffer &append)
287 {
288     // function wrapper for parent class
289     append_buffer_data(append.d_buffer);
290 }
291
292 void
293 dna_buffer::append_buffer_data(const std::vector<gr_complex> &append)
294 {
295     // after checking to see if the array passed
296     // is empty, the array data will be appended
297     // with the user defined length of empty data
298     if (!append.empty())
299     {

```

```

300     for(int i = 0; i < d_space + append.size(); i++)
301     {
302         if(i < d_space)
303         {
304             d_buffer.push_back(gr_complex(0.0f,0.0f));
305         } else{
306             //d_buffer.push_back(append[i - d_space]);
307             append_buffer_data(append[i - d_space]);
308         }
309     }
310 }
311 }
312
313 void
314 dna_buffer::append_buffer_data(const gr_complex &element)
315 {
316     d_dna_buffer_on = true; // by default, if any data is appended, then the buffer must be set
317     d_buffer.push_back(element);
318 }
319
320 void
321 dna_buffer::store_preamble_data(const int &input_size,
322                                const gr_complex *in_rx)
323 {
324
325     // initial check to ensure that the size
326     // of the received data is worth copying
327     // before estimating the indexes
328     if(input_size > 0)
329     {
330         std::vector<gr_complex> temp; // temporary placeholder
331         set_buffer();
332
333         for(int i = 0; i < input_size; i++)
334         {
335             temp.push_back(in_rx[i]);
336         }
337
338         d_buffer = std::move(temp);
339     }
340 }

```

D.4 Base Packet and IEEE 802.15.4 Packet Classes

```

1  /*****
   ***
2  *packet_base Class Definition
3  *****/
   **/
4  /*!
5  * Class: packet_base

```

```

6  * Author: Cruz, Frankie
7  * Description: Base class interface for minimalist
8  *             PHY Packet size calculation based
9  *             on on preamble section , 1 length ,
10 *             section , and 1 payload section .
11 *             Each standard breaks these three
12 *             down further into smaller subsets .
13 * Generated: Tues Nov 20 22:00:18 2018
14 * Modified: Dec 01 13:30:01 2018
15 * Build Version: 1.3
16 * Changes: --fixed get_preamble_len_samples
17 */
18 class packet_base
19 {
20     public:
21     packet_base() = default;
22     ~packet_base() = default;
23
24     /*!
25     * Setters Interface for preamble, length ,
26     * and data packet field sizes
27     */
28     virtual void set_preamble_len_bytes(const int &preamble_len){d_preamble_len = preamble_len;};
29     virtual void set_length_len_bytes(const int &length_len){d_length_len = length_len;};
30     virtual void set_data_len_bytes(const int &data_len){d_data_len = data_len;};
31
32     /*!
33     * Getters Interface for preamble, length ,
34     * and data packet field sizes
35     */
36     virtual int get_preamble_len_bytes() const {return d_preamble_len;};
37     virtual int get_length_len_bytes() const {return d_length_len;};
38     virtual int get_data_len_bytes() const {return d_data_len;};
39     virtual int get_packet_len_bytes() const {return (d_preamble_len + d_length_len + d_data_len);};
40     virtual int get_preamble_len_chips() const {return round(d_byte2sym * d_sym2chip * d_preamble_len);};
41     virtual int get_length_len_chips() const {return round(d_byte2sym * d_sym2chip * d_length_len);};
42     virtual int get_data_len_chips() const {return round(d_byte2sym * d_sym2chip * d_data_len);};
43     virtual int get_packet_len_chips() const {return round(d_byte2sym * d_sym2chip * get_packet_len_bytes());};
44     virtual int get_preamble_len_samples() const {return round(d_chip_period * d_sample_rate * get_preamble_len_chips());};
45     virtual int get_length_len_samples() const {return round(d_chip_period * d_sample_rate * get_length_len_chips());};
46     virtual int get_data_len_samples() const {return round(d_chip_period * d_sample_rate * get_data_len_chips());};
47     virtual int get_packet_len_samples() const {return round(d_chip_period * d_sample_rate * get_packet_len_chips());};
48
49     /*!
50     * Setters Interface for rates and periods
51     */
52     virtual void set_chip_period(const float &chip_period);
53     virtual void set_symbol_period(const float &symbol_period);
54     virtual void set_sample_period(const float &sample_period);
55     virtual void set_chip_rate(const float &chip_rate);
56     virtual void set_symbol_rate(const float &symbol_rate);
57     virtual void set_sample_rate(const float &sample_rate);
58
59     /*!

```

```

60     * Setters Interface for conversion multipliers
61     */
62     virtual void set_byte_to_sym_mult(const float &byte2sym);
63     virtual void set_sym_to_byte_mult(const float &sym2byte);
64     virtual void set_sym_to_chip_mult(const float &sym2chip);
65     virtual void set_chip_to_sym_mult(const float &chip2sym);
66
67     /*!
68     * Getters Interface for rates and periods
69     */
70     virtual float get_chip_period() const {return d_chip_period;};
71     virtual float get_symbol_period() const {return d_symbol_period;};
72     virtual float get_sample_period() const {return d_sample_period;};
73     virtual float get_chip_rate() const {return 1/d_chip_period;};
74     virtual float get_symbol_rate() const {return 1/d_symbol_period;};
75     virtual float get_sample_rate() const {return 1/d_sample_period;};
76
77     /*!
78     * Getters Interface for conversion multipliers
79     */
80     virtual float get_byte_to_sym_mult() const {return d_byte2sym;};
81     virtual float get_sym_to_byte_mult() const {return d_sym2byte;};
82     virtual float get_sym_to_chip_mult() const {return d_sym2chip;};
83     virtual float get_chip_to_sym_mult() const {return d_chip2sym;};
84     virtual float get_bit_to_byte_mult() const {return d_bit2byte;};
85     virtual float get_byte_to_bit_mult() const {return 1/d_bit2byte;};
86
87     private:
88     int d_preamble_len = 0; // preamble frame length in bytes
89     int d_length_len = 0; // data length field parameter in bytes
90     int d_data_len = 0; // data packet frame length in bytes
91     float d_bit2byte = 0.125f; // 1-bit is 1/8 byte (byte/bit)
92     float d_chip_period = 0.0f; // (seconds/chip)
93     float d_symbol_period = 0.0f; // (seconds/symbol)
94     float d_chip_rate = 0.0f; // (chip/sec)
95     float d_symbol_rate = 0.0f; // (sym/sec)
96     float d_byte2sym = 0.0f; // 1-byte = 2-symbols (sym/byte)
97     float d_sym2byte = 0.0f; // 1-symbol = 1/2 byte (byte/sym)
98     float d_sym2chip = 0.0f; // 1-symbol = 32-chips (chips/sym)
99     float d_chip2sym = 0.0f; // 1-chip = 1/32-symbols (sym/chip)
100    float d_sample_period = 0.0f; // (sec/sample)
101    float d_sample_rate = 0.0f; // (sample/sec)
102 };
103
104 /*****
105     ***
106     *ieee802_15_4_packet Class Definition
107     *****/
108     /*!
109     * Class: ieee802_15_4_packet
110     * Author: Cruz, Frankie
111     * Description: IEEE802-15-4 class for
112     * PHY Packet size calculation.

```

```

112 * Generated: Tues Nov 21 2:00:18 2018
113 * Modified: Wed Nov 28 10:30:11 2018
114 * Build Version: 1.2
115 * Changes: -Removed some initializing setter
116 *          -Need to improve and fix functionality
117 */
118 class ieee802_15_4_packet : public packet_base
119 {
120     public:
121
122     /*!
123      * Constructor's and Destructor
124      */
125     ieee802_15_4_packet();
126     ieee802_15_4_packet(const float &samp_rate);
127     ieee802_15_4_packet(const int &preamble_len,
128                         const int &length_len,
129                         const int &data_len,
130                         const float &samp_rate);
131     ~ieee802_15_4_packet() = default;
132
133     /*!
134      * Copy Constructor & Assignment
135      */
136     ieee802_15_4_packet(const ieee802_15_4_packet &copy);
137     ieee802_15_4_packet &operator=(const ieee802_15_4_packet &copy);
138
139     /*!
140      * Getters for bytes, chips, and samples for
141      * ieee802-15-4 packet sections
142      */
143     int get_shr_preamble_len_byte() const;
144     int get_shr_sfd_len_bytes() const;
145     int get_phr_len_bytes() const;
146     int get_payload_len_bytes() const;
147     int get_crc_len_bytes() const;
148     int get_shr_preamble_len_chips() const;
149     int get_shr_sfd_len_chips() const;
150     int get_phr_len_chips() const;
151     int get_payload_len_chips() const;
152     int get_crc_len_chips() const;
153     int get_shr_preamble_len_samples() const;
154     int get_shr_sfd_len_samples() const;
155     int get_phr_len_samples() const;
156     int get_payload_len_samples() const;
157     int get_crc_len_samples() const;
158
159     private:
160     const int d_shr_preamble_len = 4; // sync preamble (4 - 0x00 or 32-bits)
161     const int d_shr_sfd_len = 1; // start of frame delimiter (1 - 0x7A or 8-bits)
162     const int d_phr_len = 1; // phr header denoting packet length
163     const int d_payload_len = 125; // payload length (max 125-bytes or 1000-bits)
164     const int d_crc_len = 2; // crc to check packet validity
165

```

```

166 };
167
168 /*****
169     ***
170     *packet_base Source Code
171     ****
172     */
173 void
174 packet_base::set_chip_period(const float &chip_period)
175 {
176     d_chip_period = chip_period;
177
178     if (d_chip_rate != (1/chip_period))
179     {
180         d_chip_rate = 1/chip_period;
181     }
182 }
183
184 void
185 packet_base::set_symbol_period(const float &symbol_period)
186 {
187     d_symbol_period = symbol_period;
188
189     if (d_symbol_rate != (1/symbol_period))
190     {
191         d_symbol_rate = 1/d_symbol_period;
192     }
193 }
194
195 void
196 packet_base::set_sample_period(const float &sample_period)
197 {
198     d_sample_period = sample_period;
199
200     if (d_sample_rate != (1/sample_period))
201     {
202         d_sample_rate = 1/sample_period;
203     }
204 }
205
206 void
207 packet_base::set_chip_rate(const float &chip_rate)
208 {
209     d_chip_rate = chip_rate;
210
211     if (d_chip_period != (1/chip_rate))
212     {
213         d_chip_period = 1/chip_rate;
214     }
215 }
216
217 void
218 packet_base::set_symbol_rate(const float &symbol_rate)
219 {

```

```

218     d_symbol_rate = symbol_rate;
219
220     if (d_symbol_period != (1/symbol_rate))
221     {
222         d_symbol_period = 1/symbol_rate;
223     }
224 }
225
226 void
227 packet_base::set_sample_rate(const float &sample_rate)
228 {
229     d_sample_rate = sample_rate;
230
231     if (d_sample_period != (1/sample_rate))
232     {
233         d_sample_period = 1/sample_rate;
234     }
235 }
236
237 void
238 packet_base::set_byte_to_sym_mult(const float &byte2sym)
239 {
240     d_byte2sym = byte2sym;
241
242     if (d_sym2byte != (1/byte2sym))
243     {
244         d_sym2byte = 1/byte2sym;
245     }
246 }
247
248 void
249 packet_base::set_sym_to_byte_mult(const float &sym2byte)
250 {
251     d_sym2byte = sym2byte;
252
253     if (d_byte2sym != (1/sym2byte))
254     {
255         d_byte2sym = 1/sym2byte;
256     }
257 }
258
259 void
260 packet_base::set_sym_to_chip_mult(const float &sym2chip)
261 {
262     d_sym2chip = sym2chip;
263
264     if (d_chip2sym != (1/sym2chip))
265     {
266         d_chip2sym = 1/sym2chip;
267     }
268 }
269
270 void
271 packet_base::set_chip_to_sym_mult(const float &chip2sym)

```

```

272 {
273     d_chip2sym = chip2sym;
274
275     if (d_sym2chip != (1 / chip2sym))
276     {
277         d_sym2chip = 1 / chip2sym;
278     }
279 }
280
281 /*****
282     ***
283     *ieee802_15_4_packet Source Code
284     *****/
285
286 ieee802_15_4_packet::ieee802_15_4_packet()
287 {
288     set_preamble_len_bytes(d_shr_preamble_len + d_shr_sfd_len);
289     set_length_len_bytes(d_phr_len);
290     set_data_len_bytes(d_payload_len + d_crc_len);
291     set_sample_rate(4e6f);
292     set_chip_rate(2e6f);
293     set_symbol_rate(1e6f);
294     set_byte_to_sym_mult(2.0f);
295     set_sym_to_chip_mult(32.0f);
296 }
297
298 ieee802_15_4_packet::ieee802_15_4_packet(const float &samp_rate)
299 {
300     set_preamble_len_bytes(d_shr_preamble_len + d_shr_sfd_len);
301     set_length_len_bytes(d_phr_len);
302     set_data_len_bytes(d_payload_len + d_crc_len);
303     set_sample_rate(samp_rate);
304     set_chip_rate(2e6f);
305     set_symbol_rate(1e6f);
306     set_byte_to_sym_mult(2.0f);
307     set_sym_to_chip_mult(32.0f);
308 }
309
310 ieee802_15_4_packet::ieee802_15_4_packet(const int &preamble_len,
311                                         const int &length_len,
312                                         const int &data_len,
313                                         const float &samp_rate)
314 {
315     set_preamble_len_bytes(preamble_len);
316     set_length_len_bytes(length_len);
317     set_data_len_bytes(data_len);
318     set_sample_rate(samp_rate);
319     set_chip_rate(2e6f);
320     set_symbol_rate(1e6f);
321     set_byte_to_sym_mult(2.0f);
322     set_sym_to_chip_mult(32.0f);
323 }
324
325 ieee802_15_4_packet::ieee802_15_4_packet(const ieee802_15_4_packet &copy)

```

```

324 {
325     set_preamble_len_bytes(copy.get_preamble_len_bytes());
326     set_length_len_bytes(copy.get_length_len_bytes());
327     set_data_len_bytes(copy.get_data_len_bytes());
328     set_sample_period(copy.get_sample_period());
329     set_sample_rate(copy.get_sample_rate());
330     set_chip_rate(copy.get_chip_rate());
331     set_chip_period(copy.get_chip_period());
332     set_symbol_period(copy.get_symbol_period());
333     set_symbol_rate(copy.get_symbol_rate());
334     set_byte_to_sym_mult(copy.get_byte_to_sym_mult());
335     set_sym_to_byte_mult(copy.get_sym_to_byte_mult());
336     set_sym_to_chip_mult(copy.get_sym_to_chip_mult());
337     set_chip_to_sym_mult(copy.get_chip_to_sym_mult());
338 }
339
340 ieee802_15_4_packet
341 &ieee802_15_4_packet::operator=(const ieee802_15_4_packet &copy)
342 {
343     //self assignment check
344     if(&copy != this)
345     {
346         set_preamble_len_bytes(copy.get_preamble_len_bytes());
347         set_length_len_bytes(copy.get_length_len_bytes());
348         set_data_len_bytes(copy.get_data_len_bytes());
349         set_sample_period(copy.get_sample_period());
350         set_sample_rate(copy.get_sample_rate());
351         set_chip_rate(copy.get_chip_rate());
352         set_chip_period(copy.get_chip_period());
353         set_symbol_period(copy.get_symbol_period());
354         set_symbol_rate(copy.get_symbol_rate());
355         set_byte_to_sym_mult(copy.get_byte_to_sym_mult());
356         set_sym_to_byte_mult(copy.get_sym_to_byte_mult());
357         set_sym_to_chip_mult(copy.get_sym_to_chip_mult());
358         set_chip_to_sym_mult(copy.get_chip_to_sym_mult());
359     }
360
361     return *this;
362 }
363
364 int
365 ieee802_15_4_packet::get_shr_preamble_len_byte() const
366 {
367     return d_shr_preamble_len;
368 }
369
370 int
371 ieee802_15_4_packet::get_shr_sfd_len_bytes() const
372 {
373     return d_shr_sfd_len;
374 }
375
376 int
377 ieee802_15_4_packet::get_phr_len_bytes() const

```

```

378 {
379     return d_phr_len;
380 }
381
382 int
383 ieee802_15_4_packet::get_payload_len_bytes() const
384 {
385     return d_payload_len;
386 }
387
388 int
389 ieee802_15_4_packet::get_crc_len_bytes() const
390 {
391     return d_crc_len;
392 }
393
394 int
395 ieee802_15_4_packet::get_shr_preamble_len_chips() const
396 {
397     return round(d_shr_preamble_len *
398                 get_byte_to_sym_mult() *
399                 get_sym_to_chip_mult());
400 }
401
402 int
403 ieee802_15_4_packet::get_shr_sfd_len_chips() const
404 {
405     return round(d_shr_sfd_len *
406                 get_byte_to_sym_mult() *
407                 get_sym_to_chip_mult());
408 }
409
410 int
411 ieee802_15_4_packet::get_phr_len_chips() const
412 {
413     return round(d_phr_len *
414                 get_byte_to_sym_mult() *
415                 get_sym_to_chip_mult());
416 }
417
418 int
419 ieee802_15_4_packet::get_payload_len_chips() const
420 {
421     return round(d_payload_len *
422                 get_byte_to_sym_mult() *
423                 get_sym_to_chip_mult());
424 }
425
426 int
427 ieee802_15_4_packet::get_crc_len_chips() const
428 {
429     return round(d_crc_len *
430                 get_byte_to_sym_mult() *
431                 get_sym_to_chip_mult());

```

```

432 }
433
434 int
435 ieee802_15_4_packet::get_shr_preamble_len_samples() const
436 {
437     return round(get_shr_preamble_len_chips() *
438                 get_chip_period() *
439                 get_sample_rate());
440 }
441
442 int
443 ieee802_15_4_packet::get_shr_sfd_len_samples() const
444 {
445     return round(get_shr_sfd_len_chips() *
446                 get_chip_period() *
447                 get_sample_rate());
448 }
449
450 int
451 ieee802_15_4_packet::get_phr_len_samples() const
452 {
453     return round(get_phr_len_chips() *
454                 get_chip_period() *
455                 get_sample_rate());
456 }
457
458 int
459 ieee802_15_4_packet::get_payload_len_samples() const
460 {
461     return round(get_payload_len_chips() *
462                 get_chip_period() *
463                 get_sample_rate());
464 }
465
466 int
467 ieee802_15_4_packet::get_crc_len_samples() const
468 {
469     return round(get_crc_len_chips() *
470                 get_chip_period() *
471                 get_sample_rate());
472 }

```

D.5 File Read Implementation Class

```

1  /*****
    ***
2  *file_read_impl Class Definition
3  *****/
   **/
4  /*!
5  * Class: file_read_impl

```

```

6  * Author: GNURadio; Cruz, Frankie
7  * Description: Based off of file_source_impl
8  *             hardcoded due to time constraints.
9  *             Essentially load binary data
10 * Generated: Thurs Dec 06 11:00:00 2018
11 * Modified: Thurs Dec 07 10:00:00 2018
12 * Build Version: 1.1
13 * Changes: -added a string store & getter function
14 *           for the filename
15 *           -added getter for itemsize
16 */
17 class file_read_impl
18 {
19     private:
20         size_t d_itemsize;
21         FILE *d_fp;
22         FILE *d_new_fp;
23         bool d_repeat;
24         bool d_updated;
25         bool d_file_begin;
26         long d_repeat_cnt;
27         pmt::pmt_t d_add_begin_tag;
28         std::string d_filename; //storage for filename
29
30         boost::mutex fp_mutex;
31         pmt::pmt_t _id;
32
33         void do_update();
34
35     public:
36         file_read_impl(const size_t &itemsize, const char *filename, const bool &repeat = false);
37         ~file_read_impl();
38
39         bool seek(long seek_point, int whence);
40         void open(const char *filename, bool repeat);
41         void close();
42
43         int read_data(const int &array_size,
44                     void* read_data);
45
46         void set_begin_tag(const pmt::pmt_t &val);
47         std::string get_filename() const;
48         size_t get_size() const;
49
50 };
51
52 /*****
53  ***
54  *file_read_impl Source Code
55  *****/
56
57 file_read_impl::file_read_impl(const size_t &itemsize, const char *filename, const bool &repeat)
58     : d_itemsize(itemsize), d_fp(0), d_new_fp(0),
59     d_repeat(repeat), d_updated(false), d_file_begin(true),

```

```

58             d_repeat_cnt(0), d_add_begin_tag(pmt::PMT_NIL)
59     {
60         open(filename, repeat);
61         do_update();
62         d_filename.append(filename);
63
64         std::stringstream str;
65         //str << name() << unique_id();
66         str << "AFIT File Read!!!";
67         _id = pmt::string_to_symbol(str.str());
68     }
69
70     file_read_impl::~file_read_impl()
71     {
72         if (d_fp)
73             fclose ((FILE*)d_fp);
74         if (d_new_fp)
75             fclose ((FILE*)d_new_fp);
76     }
77
78     bool
79     file_read_impl::seek(long seek_point, int whence)
80     {
81         return fseek((FILE*)d_fp, seek_point * d_itemsize, whence) == 0;
82     }
83
84
85     void
86     file_read_impl::open(const char *filename, bool repeat)
87     {
88         // obtain exclusive access for duration of this function
89         gr::thread::scoped_lock lock(fp_mutex);
90
91         int fd;
92
93         // we use "open" to use to the O_LARGEFILE flag
94         if ((fd = ::open(filename, O_RDONLY | OUR_O_LARGEFILE | OUR_O_BINARY)) < 0)
95         {
96             perror(filename);
97             throw std::runtime_error("can't open file");
98         }
99
100        if (d_new_fp)
101        {
102            fclose(d_new_fp);
103            d_new_fp = 0;
104        }
105
106        if ((d_new_fp = fdopen (fd, "rb")) == NULL)
107        {
108            perror(filename);
109            ::close(fd); // don't leak file descriptor if fdopen fails
110            throw std::runtime_error("can't open file");
111        }

```

```

112
113     //Check to ensure the file will be consumed according to item size
114     fseek(d_new_fp, 0, SEEK_END);
115     int file_size = ftell(d_new_fp);
116     rewind (d_new_fp);
117
118     //Warn the user if part of the file will not be consumed.
119     if (file_size % d_itemsize){
120         GR_LOG_WARN(d_logger, "WARNING: File will not be fully consumed with the current output type");
121     }
122
123     d_updated = true;
124     d_repeat = repeat;
125 }
126
127 void
128 file_read_impl::close()
129 {
130     // obtain exclusive access for duration of this function
131     gr::thread::scoped_lock lock(fp_mutex);
132
133     if (d_new_fp != NULL)
134     {
135         fclose(d_new_fp);
136         d_new_fp = NULL;
137     }
138     d_updated = true;
139 }
140
141 void
142 file_read_impl::do_update()
143 {
144     if (d_updated)
145     {
146         gr::thread::scoped_lock lock(fp_mutex); // hold while in scope
147
148         if (d_fp)
149             fclose(d_fp);
150
151         d_fp = d_new_fp;    // install new file pointer
152         d_new_fp = 0;
153         d_updated = false;
154         d_file_begin = true;
155     }
156 }
157
158 void
159 file_read_impl::set_begin_tag(const pmt::pmt_t &val)
160 {
161     d_add_begin_tag = val;
162 }
163
164 int
165 file_read_impl::read_data(const int &array_size,

```

```

166             void *read_data)
167     {
168         char *o = (char*)read_data;
169         int i;
170         int size = array_size;
171
172         do_update(); // update d_fp is reqd
173         if(d_fp == NULL)
174             throw std::runtime_error("work with file not open");
175
176         gr::thread::scoped_lock lock(fp_mutex); // hold for the rest of this function
177
178         while(size) {
179             i = fread(o, d_itemsize, size, (FILE*)d_fp);
180
181             size -= i;
182             o += i * d_itemsize;
183
184             if(size == 0) // done
185                 break;
186
187             if(i > 0) // short read, try again
188                 continue;
189
190             // We got a zero from fread. This is either EOF or error. In
191             // any event, if we're in repeat mode, seek back to the beginning
192             // of the file and try again, else break
193             if(!d_repeat)
194                 break;
195
196             if(fseek((FILE *) d_fp, 0, SEEK_SET) == -1) {
197                 fprintf(stderr, "[%s] fseek failed\n", __FILE__);
198                 exit(-1);
199             }
200
201             if (d_add_begin_tag != pmt::PMT_NIL)
202             {
203                 d_file_begin = true;
204                 d_repeat_cnt++;
205             }
206         }
207
208         if(size > 0) // EOF or error
209         {
210             if(size == array_size) // we didn't read anything; say we're done
211                 return -1;
212             return array_size - size; // else return partial result
213         }
214
215         return array_size;
216     }
217
218     std::string
219     file_read_impl::get_filename() const{

```

```

220     return d_filename;
221 }
222
223 size_t
224 file_read_impl::get_size() const
225 {
226     return d_itemsize;
227 }

```

D.6 File Store Base and File Store Implementation Classes

```

1  /*****
    ***
2  *file_store_base Class Definition
3  *****/
    **
4  /*!
5   * Class: file_store_base
6   * Author: GNURadio; Cruz, Frankie
7   * Description: Based off of file_sink_base
8   *             hardcoded due to time constraints
9   * Generated: Thurs Dec 06 11:00:00 2018
10  * Modified: Thurs Dec 07 10:00:00 2018
11  * Build Version: 1.1
12  * Changes: -added a string and getter function
13  *          for the filename
14  */
15  class file_store_base
16  {
17  protected:
18      FILE      *d_fp;          // current FILE pointer
19      FILE      *d_new_fp;     // new FILE pointer
20      bool      d_updated;     // is there a new FILE pointer?
21      bool      d_is_binary;
22      boost::mutex d_mutex;
23      bool      d_unbuffered;
24      bool      d_append;
25      std::string d_filename; // c_string of filename
26
27  protected:
28      file_store_base(const char *filename, const bool &is_binary, const bool &append);
29
30  public:
31      file_store_base() {}
32      ~file_store_base();
33
34      /*!
35       * \brief Open filename and begin output to it.
36       */
37      bool open(const char *filename);
38

```

```

39     /*!
40     * \brief Close current output file .
41     *
42     * Closes current output file and ignores any output until
43     * open is called to connect to another file .
44     */
45     void close();
46
47     /*!
48     * \brief if we've had an update, do it now.
49     */
50     void do_update();
51
52     /*!
53     * \brief turn on unbuffered writes for slower outputs
54     */
55     void set_unbuffered(const bool &unbuffered);
56
57     /*!
58     * option to append data to file
59     */
60     void set_append(const bool &append);
61
62     std::string get_filename() const; // returns stored filename
63
64 };
65
66 /*****
67     ***
68     *file_store_impl Class Definition
69     *****/
70
71 /*!
72 * Class: file_store_impl
73 * Author: GNURadio; Cruz, Frankie
74 * Description: Based off of file_sink_impl
75 *              hardcoded due to time constraints.
76 *              Essentially save's binary data
77 * Generated: Thurs Dec 06 11:00:00 2018
78 * Modified: Thurs Dec 06 11:00:00 2018
79 * Build Version: 1.0
80 * Changes: -Baseline
81 */
82
83 class file_store_impl : public file_store_base
84 {
85 private:
86     size_t d_itemsize;
87 public:
88     //file_read_impl will take the item_size using the sizeof() function of an int, float, gr_complex,
89     // or char.
90     file_store_impl(const size_t &itemsz, const char *filename,
91                     const bool &binary_data = true, const bool &append = false);
92     //file_store_impl() {}

```

```

91     ~file_store_impl();
92
93     int store_data(const int &array_size,
94                   void* input_items);
95
96     size_t get_size() const; // returns stored filesize
97 };
98
99 /*****
100  ***
101  *file_store_base Source Code
102  *****/
103
104 file_store_base::file_store_base(const char *filename, const bool &is_binary, const bool &append)
105 : d_fp(0), d_new_fp(0), d_updated(false), d_is_binary(is_binary), d_append(append)
106 {
107     if (!open(filename))
108         throw std::runtime_error ("can't open file");
109     d_filename.append(filename);
110 }
111
112 file_store_base::~file_store_base()
113 {
114     close();
115     if (d_fp) {
116         fclose(d_fp);
117         d_fp = 0;
118     }
119 }
120
121 bool
122 file_store_base::open(const char *filename)
123 {
124     gr::thread::scoped_lock guard(d_mutex); // hold mutex for duration of this function
125
126     // we use the open system call to get access to the O_LARGEFILE flag.
127     int fd;
128     int flags;
129     if (d_append) {
130         flags = O_WRONLY|O_CREAT|O_APPEND|OUR_O_LARGEFILE|OUR_O_BINARY;
131     } else {
132         flags = O_WRONLY|O_CREAT|O_TRUNC|OUR_O_LARGEFILE|OUR_O_BINARY;
133     }
134     if ((fd = ::open(filename, flags, 0664)) < 0) {
135         perror(filename);
136         return false;
137     }
138     if (d_new_fp) { // if we've already got a new one open, close it
139         fclose(d_new_fp);
140         d_new_fp = 0;
141     }
142     if ((d_new_fp = fdopen (fd, d_is_binary ? "wb" : "w")) == NULL) {
143         perror (filename);

```

```

143     ::close(fd);          // don't leak file descriptor if fdopen fails.
144 }
145
146     d_updated = true;
147     return d_new_fp != 0;
148 }
149
150 void
151 file_store_base::close()
152 {
153     gr::thread::scoped_lock guard(d_mutex);    // hold mutex for duration of this function
154
155     if(d_new_fp) {
156         fclose(d_new_fp);
157         d_new_fp = 0;
158     }
159     d_updated = true;
160 }
161
162 void
163 file_store_base::do_update()
164 {
165     if(d_updated) {
166         gr::thread::scoped_lock guard(d_mutex);    // hold mutex for duration of this block
167         if(d_fp)
168             fclose(d_fp);
169         d_fp = d_new_fp;                // install new file pointer
170         d_new_fp = 0;
171         d_updated = false;
172     }
173 }
174
175 void
176 file_store_base::set_unbuffered(const bool &unbuffered)
177 {
178     d_unbuffered = unbuffered;
179 }
180
181 void
182 file_store_base::set_append(const bool &append)
183 {
184     d_append = append;
185 }
186
187 std::string
188 file_store_base::get_filename() const
189 {
190     return d_filename;
191 }
192
193 /*****
194     ***
195     *file_store_impl Source Code

```

```

195  ****
196  file_store_impl::file_store_impl(const size_t &itemsize, const char *filename,
197                                  const bool &binary_data, const bool &append)
198                                  : file_store_base(filename, binary_data, append),
199                                  d_itemsize(itemsize)
200  {
201  }
202
203  file_store_impl::~file_store_impl()
204  {
205  }
206
207  int
208  file_store_impl::store_data(const int &array_size,
209                              void* input_items)
210  {
211      char *inbuf = (char*) input_items;
212      int  nwritten = 0;
213
214      do_update();          // update d_fp is reqd
215
216      if (!d_fp)
217          return array_size;    // drop output on the floor
218
219      while(nwritten < array_size) {
220          int count = fwrite(inbuf, d_itemsize, array_size - nwritten, d_fp);
221          if (count == 0) {
222              if (ferror(d_fp)) {
223                  std::stringstream s;
224                  s << "file_sink write failed with error " << fileno(d_fp) << std::endl;
225                  throw std::runtime_error(s.str());
226              }
227              else { // is EOF
228                  break;
229              }
230          }
231          nwritten += count;
232          inbuf += count * d_itemsize;
233      }
234
235      if (d_unbuffered)
236          fflush (d_fp);
237
238      return nwritten;
239  }
240
241  size_t
242  file_store_impl::get_size() const
243  {
244      return d_itemsize;
245  }

```

Bibliography

1. G. Shi and K. Li, "Signal Interference in WiFi and ZigBee Networks." New York City: Springer Publishing, 2017, ch. 2, pp. 9–28. [Online]. Available: <http://link.springer.com/10.1007/978-3-319-47806-7>
2. F. Eichelberger, "Using Software Defined Radio to Attack "Smart Home" Systems," SANS Institute, Tech. Rep., 2014.
3. B. W. Ramsey, B. E. Mullins, W. M. Lowder, and R. M. Speers, "Sharpening the Stinger: Tuning KillerBee for Critical Infrastructure Warwalking," in *IEEE Military Communications Conference*, oct 2014, pp. 104–109.
4. R. L. Security, "Killerbee IEEE 802.15.4 Research Tool." [Online]. Available: <https://github.com/riverloopsec/killerbee>; <http://www.riverloopsecurity.com/>
5. D. W. Benjamin Ramsey, "Improved Wireless Security Through Physical Layer Protocol Manipulation and Radio Frequency Fingerprinting."
6. A. Betances, K. M. Hopkinson, and M. D. Silvius, "Detection of Primary User Emulation Attacks Using Constellation-Based Distinct Native Attribute Techniques," 2016.
7. T. J. Carbino, M. A. Temple, and J. Lopez, "Conditional Constellation Based-Distinct Native Attribute (CB-DNA) Fingerprinting for Network Device Authentication," in *IEEE International Conference on Communications (ICC)*, 2016.
8. C. W. Coon, "Comparative Analysis of RF Emission Based Fingerprinting Techniques for ZigBee Device Classification, AFIT-ENG-MS-17-M-017," Ph.D. dissertation, Masters Thesis, Air Force Institute of Technology, 2017.
9. C. Dubendorfer, "Using RF-DNA Fingerprints to Discriminate ZIGBEE Devices in an Operational Environment," Ph.D. dissertation, Masters Thesis, Air Force Institute of Technology, 2013.
10. W. M. Lowder, "Real-Time RF-DNA Fingerprinting of ZigBee Devices Using A Software-Defined Radio with FPGA Processing, AFIT-ENG-MS-15-M-054," Ph.D. dissertation, Masters Thesis, Air Force Institute of Technology, 2015.
11. M. Lukacs, P. Collins, and M. Temple, "Classification performance using RF-DNA fingerprinting of ultra-wideband noise waveforms," *Electronics Letters*, vol. 51, no. 10, pp. 787–789, 2015.
12. H. Patel, M. A. Temple, and B. W. Ramsey, "Comparison of High-end and Low-end Receivers for RF-DNA Fingerprinting," in *2014 IEEE Military Communications Conference*, oct 2014, pp. 24–29.

13. C. M. Rondeau, J. A. Betances, and M. A. Temple, "Securing ZigBee Commercial Communications Using Constellation Based Distinct Native Attribute Fingerprinting," *Security and Communication Networks*, vol. 2018, 2018.
14. M. A. Temple, "EENG 699 - RF Fingerprinting," p. 22, 2018.
15. D. J. Timothy Carbino, "Exploitation of Unintentional Ethernet Cable Emissions Using Constellation Based-Distinct Native Attribute (CB-DNA) Fingerprints To Enhance Network Security," 2015.
16. C. K. Dubendorfer, B. W. Ramsey, and M. A. Temple, "An RF-DNA verification process for ZigBee networks," in *MILCOM 2012 - 2012 IEEE Military Communications Conference*, oct 2012, pp. 1–6.
17. M. Lukacs, P. Collins, and M. Temple, "Device identification using active noise interrogation and RF-DNA "fingerprinting" for non-destructive amplifier acceptance testing," in *IEEE 17th Annual Wireless and Microwave Technology Conference (WAMICON)*, apr 2016, pp. 1–6.
18. D. R. Reising, M. A. Temple, and M. E. Oxley, "Gabor-based RF-DNA fingerprinting for classifying 802.16e WiMAX Mobile Subscribers," in *International Conference on Computing, Networking and Communications (ICNC)*, jan 2012, pp. 7–13.
19. B. W. Ramsey, B. E. Mullins, M. A. Temple, and M. R. Grimaila, "Wireless Intrusion Detection and Device Fingerprinting through Preamble Manipulation," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 5, pp. 585–596, sep 2015.
20. Z. Alliance, "Zigbee Specification," p. 622, 2012. [Online]. Available: <http://www.zigbee.org/>
21. T. W. Foundation, "Wireshark." [Online]. Available: <https://www.wireshark.org/>
22. T. G. R. Foundation, "GNURadio." [Online]. Available: https://wiki.gnuradio.org/index.php/Main_Page;<https://www.gnuradio.org/doc/doxygen/index.html>;<https://wiki.gnuradio.org/index.php/Download>;<https://github.com/gnuradio/gnuradio>
23. I. C. Society, *IEEE Standard for Local and metropolitan Area Networks IEEE 802.15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)*. New York City: The Institute of Electrical and Electronics Engineers, Inc, 2011, no. September.
24. T. Schmid, "GNU Radio 802.15.4 En- and Decoding," University of California, Los Angeles, Tech. Rep., 2007.
25. B. Sklar, *Digital Communications Fundamentals and Applications*, 2nd ed. Pearson Education, 2016.

26. S. Proakis, John G; Masoud, *Digital Communications*, 5th ed. New York City: McGraw-Hill Higher Education, 2008.
27. Reising, Donald, "Exploitation of RF-DNA for Device Classification and Verification Using GRLVQI Processing," 2012.
28. J. C. Milton, J. Susan; Arnold, *Introduction to Probability and Statistics*, 4th ed., McGraw-Hill, Ed., New York City, 2003.
29. R. Electronics, "Ramsey STE4400," 2019. [Online]. Available: <http://www.ramseyelectronics.com/product.php?pid=17>
30. M. Haykin, Simon; Moher, *Communication Systems*, 5th ed. Hoboken: Wiley Publishing, 2009.
31. B. G. Jacob, "Eigen." [Online]. Available: http://eigen.tuxfamily.org/index.php?title=Main_Page; <http://eigen.tuxfamily.org/dox/GettingStarted.html>
32. A. M. Collins, Travis F; Getz, Robin; Pu, Di; Wyglinski, *Software-Defined Radio for Engineers*, 1st ed. Norwood: Artech House, 2018. [Online]. Available: www.artechhouse.com
33. E. R. A. N. I. Corporation, "Ettus Research." [Online]. Available: <https://www.ettus.com/product/details/USRP-B205mini-i-Board>; [https://kb.ettus.com/Building_and_Installing_the_USRP_Open-Source_Toolchain_\(UHD_and_GNU_Radio\)_on_Linux](https://kb.ettus.com/Building_and_Installing_the_USRP_Open-Source_Toolchain_(UHD_and_GNU_Radio)_on_Linux)
34. B. Bloessl, "gr-ieee802-15-4 GNURadio Block." [Online]. Available: <https://github.com/bastibl/gr-ieee802-15-4>
35. B. Bloessl, C. Leitner, F. Dressler, and C. Sommer, "A GNU Radio-based IEEE 802.15.4 testbed," *Proceedings of 12. GI/ITG KuVS Fachgespräch Drahtlose Sensor-netze (FGSN 2013)*, pp. 37–40, 2013.
36. U. C. Workgroup, "Universal Serial Bus 3.1 Legacy Connectors and Cable Assemblies Compliance Document," p. 56, 2018. [Online]. Available: https://www.usb.org/sites/default/files/documents/cabconn_legacy_3_1_compliance_rev_1_1.pdf
37. B. Bloessl, "gr-foo GNURadio Block." [Online]. Available: <https://github.com/bastibl/gr-foo>

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE (DD-MM-YYYY) 21-03-2019		2. REPORT TYPE Master's Thesis		3. DATES COVERED (From — To) May 2017 — March 2019	
4. TITLE AND SUBTITLE Near Real-Time RF-DNA Fingerprinting for ZigBee Devices Using Software Defined Radios				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S) Cruz, Frankie A., Capt				5d. PROJECT NUMBER JON# 19G211	
				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way WPAFB OH 45433-7765				8. PERFORMING ORGANIZATION REPORT NUMBER AFIT-ENG-MS-19-M-021	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Lab, AFMC Attn: Dr. Vasu Chakravarthy 2241 Avionics Circle, Bldg 620 Wright Patterson AFB OH 45433-7765 Email: Vasu.Chakravarthy@us.af.mil, Comm: 937-713-4026				10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/RWYE	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION / AVAILABILITY STATEMENT DISTRIBUTION STATEMENT A. APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.					
13. SUPPLEMENTARY NOTES This work is declared a work of the U.S. Government and is not subject to copyright protection in the United States.					
14. ABSTRACT Low-Rate Wireless Personal Area Network(s) (LR-WPAN) usage has increased as more consumers embrace Internet of Things (IoT) devices. ZigBee Physical Layer (PHY) is based on the Institute of Electrical and Electronics Engineers (IEEE) 802.15.4 specification designed to provide a low-cost, low-power, and low-complexity solution for Wireless Sensor Network(s) (WSN). The standard's extended battery life and reliability makes ZigBee WSN a popular choice for home automation, transportation, traffic management, Industrial Control Systems (ICS), and cyber-physical systems. As robust and versatile as the standard is, ZigBee remains vulnerable to a myriad of common network attacks. Previous research involving Radio Frequency-Distinct Native Attribute (RF-DNA) Fingerprinting and device discrimination has shown that bit-level WSN security can be augmented with PHY-based features. The objective of this research was to develop and implement an Radio Frequency (RF) air monitor system that classifies devices in Near Real-Time (NRT). The performance of the NRT air monitor is contrasted against previous research that utilized MATLAB-based Fingerprinting post-processing RF-DNA. The RF air monitor demonstration included collection of IEEE 802.15.4 bursts from $N_d = 10$ RZUSBsticks to assess NRT performance and effectiveness. The first set of experiments examined how well the air monitor recovered IEEE 802.15.4 data packets while fingerprinting and discriminating ZigBee devices under two distinct workloads. The second set of experiments compared predictive post-processed MATLAB RF-DNA Multiple Discriminant Analysis/Maximum Likelihood (MDA/ML) models Average Percent Correct Classification (%C) against the air monitor's observed operational %C for each RZUSBstick. The air monitor achieved an Overall Accurate Packet Reconstruction Percent (%R) $\geq 97.92\%$ while correctly fingerprinting an Overall Fingerprinted Percent (%F) $\geq 97.48\%$ of the transmitted IEEE 802.15.4 data packets during the trials. The air monitor achieved an overall operational %C $\approx 96.93\%$ at a collected Signal-to-Noise Ratio (SNR) ≈ 33.571 dB, classified each RZUSBstick within $0.45 \text{ msec} \leq T_{MDA} \leq 1.5 \text{ msec}$ after detection, and %C Deviation (%C _Δ) = 2.71% from the collected post-processed MDA/ML model. The results support that an RF air monitor is feasible, can be effective, and will accurately operate within predictive post-processed MATLAB model estimations.					
15. SUBJECT TERMS RF-DNA, PHY, Device Classification, ZigBee					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Maj J. Addison Betances, AFIT/ENG
U	U	U	U	197	19b. TELEPHONE NUMBER (include area code) (937) 255-3636 x3305;jbetance@afit.edu