Wireless Device Key Generation

A Major Qualifying Project Report
submitted to the faculty of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science in
Electrical and Computer Engineering

By:

William Edor

Caitlyn Marcoux

Submitted to:

Project Advisor: Professor Lifeng Lai

Submitted on:

March 12, 2016

Abstract

Security is a very important concern in today’s society and is becoming even more prominent with the number of security breaches increasing rapidly. With each new encoding scheme that is developed to protect our information, an even more robust decryption scheme is also developed and is used to intercept sensitive information. While securing communications is difficult to accomplish in all mediums, securing wireless communications is the most difficult to achieve due to the broadcast nature of wireless communications and the randomness of wireless channels. Instead of viewing this as a disadvantage, recently developed physical layer security approaches argue that the channel randomness can be exploited and utilized to ensure secure communications. For example, the wireless channel between any two communicating users can be used to generate a secret key that only they will know, which will then be used to encrypt and decrypt messages. At the same time, it would be nearly impossible for any other user trying to eavesdrop on their communications, to generate this same key because of the inherent differences in channel properties. This approach for wireless key generation is thoroughly discussed in this project as well as implemented and tested using various scenarios to prove that the communication over the wireless medium is secure.

Acknowledgements

We would like to acknowledge the great assistance of Professor Lifeng Lai in the success of this project. We deeply appreciate his intellectual guidance throughout this project. We would also like to thank Edward Burnham of the Electrical and Computer Engineering Department at WPI for his immense technical help and support in setting up the logistics for the project.

Table of Contents

Abstract 2

Acknowledgements 3

Table of Figures 6

Table of Tables 7

Executive Summary 8

1. Introduction 10

  1.1: Wireless Communications 10

  1.2: Importance of Securing Wireless Communications 12

    1.2.1: TJX Security Breach 13

    1.2.2: Businesses Hacked by Wardriving Wi-Fi Networks 14

  1.3: Two Types of Security Approaches 14

    1.3.1: Computational Security 15

    1.3.2: Information Theoretic Security 16

  1.4: Our Approach 17

2. Transceiver Options 19

  2.1: Received Signal Strength Indicator (RSSI) 19

  2.2: Platform Selection 20

    2.2.1: Sensor Selection 20

    2.2.2: Software Defined Radio Selection 23

    2.2.3: Final Platform Selection 24

  2.3: Implementation Platform 25

    2.3.1: Operating System 25

    2.3.2: Programming APIs in TinyOS 26

    2.3.3: nesC Language 27

    2.3.4: Structure of a nesC Application 27

    2.3.5: nesC vs C Language 27

3. Implementation 28

  3.1: TinyOS Installation 28

  3.2: Sensor Setup 28

  3.3: Channel Estimation 30

  3.4: RSSI Quantization 33

  3.5: Error Correction 34

3.5.1: (5, 2) Binary Linear Block Channel Coding and Encoding 34
3.5.2: (5, 2) Binary Linear Block Channel Decoding 36
4. Results and Discussion 38
4.1: Scenario 1: Alice and Bob Remain Stationary 38
4.1.1: Raw RSSI Measurements 38
4.1.2: RSSI Quantization 39
4.1.3: Bit Sequence after Coding 41
4.1.4: Bit Sequence after Decoding 41
4.2: Scenario 2: Alice moving and Bob Stationary 42
4.2.1: Raw RSSI Measurements 42
4.2.2: RSSI Quantization 43
4.2.3: Bit Sequence after Coding 45
4.2.4: Bit Sequence after Decoding 46
4.3: Scenario 3: Alice, Bob, and Eve Remain Stationary 47
4.3.1: Raw RSSI Measurements 47
4.3.2: RSSI Quantization 49
4.3.3: Bit Sequence after Coding 53
4.3.4: Bit Sequence after Decoding 53
4.4: Scenario 4: Alice and Bob Remain Stationary While Eve is Moving 54
4.4.1: Raw RSSI Measurements 54
4.4.2: RSSI Quantization 55
4.4.3: Bit Sequence after Coding 60
4.4.4: Bit Sequence after Decoding 60
4.5: Testing Error Correction on Eve’s Sequence 61
4.6: Transmission Duration and Key Rate 62
5. Conclusion and Future Work 63
References 65
Appendix A 70
Appendix B 75

Table of Figures

Figure 1- Graph of Combined Wireless Service and Equipment Revenues 10
Figure 2 - Orientation of two communicating parties, Alice and Bob, and eavesdropper, Eve 17
Figure 3 - TMote Sky (CM3000) sensor 22
Figure 4 - Software-defined radio (USRP 2) 24
Figure 5 - Sensor setup for testing between Alice and Bob 29
Figure 6 - Sensor setup for Alice (laptop) moving and Bob fixed 29
Figure 7 - Sensor setup for Alice and Bob fixed while Eve (laptop) is moving and trying to listen in on communications 30
Figure 8 - State diagram of message transmission 33
Figure 9 - State diagram of message reception 33
Figure 10 - Received RSSI values at Alice and Bob before quantization 38
Figure 11 - Quantized RSSI values at Alice and Bob 39
Figure 12 - Difference between Alice and Bob's quantized sequences 40
Figure 13 - Quantized RSSI values at Alice and Bob after error correction 41
Figure 14 - Difference between Alice and Bob's quantized sequence after error correction 42
Figure 15 - Received RSSI values at Alice and Bob before quantization 43
Figure 16 - Quantized RSSI values at Alice and Bob 44
Figure 17 - Difference between Alice and Bob's quantized sequences 44
Figure 18 - Quantized RSSI values at Alice and Bob after error correction 46
Figure 19 - Difference between Alice and Bob's quantized sequence after error correction 47
Figure 20 - Received RSSI values at Alice, Bob, and Eve before quantization 48
Figure 21 - Quantized RSSI values at Alice, Bob, and Eve 49
Figure 22 - Difference between Alice and Bob's quantized sequences 50
Figure 23 - Difference between Alice and Eve's quantized sequences 51
Figure 24 - Difference between Bob and Eve's quantized sequences 52
Figure 25 - Quantized RSSI values at Alice and Bob after error correction 53
Figure 26 - Difference between Alice and Bob's quantized sequences 54
Figure 27 - Received RSSI values at Alice, Bob, and Eve before quantization 55
Figure 28 - Quantized RSSI values at Alice, Bob, and Eve 56
Figure 29 - Difference between Alice and Bob's quantized sequences 57
Figure 30 - Difference between Alice and Eve's quantized sequences 58
Figure 31 - Difference between Bob and Eve's quantized sequences 59
Figure 32 - Quantized RSSI values at Alice and Bob after error correction 60
Figure 33 - Difference between Alice and Bob's quantized sequences 61

Table of Tables

Table 1 - Breakdown of encryption utilized in a global wireless network sample population 13
Table 2 - Comparison of sensors and their characteristics 21
Table 3 - Continuation of table 2 21
Table 4 - Syndrome table [57] 37
Table 5 - Duration of transmission times for each set of 100 messages and the average between the three tests 62

Executive Summary

Security is a major concern in regards to safety, identity theft, and communications. If systems are compromised, the consequences are serious. Due to this increasing concern, this project focuses on one of these major areas of security, communications, and tests an increasingly popular method that will secure communications over a wireless channel.

There are two main ways to transmit information, over a wired connection or wirelessly. Of the two, wireless communications is becoming more popular because of the convenience brought by its mobility. Wireless connections are becoming more pronounced and more businesses, homes, and even some vehicles are supporting this advancement by supplying hotspots so wireless devices can always stay connected. While wireless communications have many positive attributes, there is one major concern. Out of the two kinds of ways to communicate, wireless communications is the more difficult to secure. This is because wireless signals are being transmitted across the air and can be intercepted by anyone in the communication range.

In order to address this problem, in this project a fairly new method for securing wireless communications was explored. This method exploits the randomness of the wireless channel to generate a secret key between two users. This key would then be used to encrypt and decrypt messages at both ends of communications. The idea behind this method is that the channel between two communicating users is the same, however, the channel of a third user who is trying to intercept these messages is different because this user would be listening from a different location and hence a different channel. In this project it is assumed that the eavesdropper is passive (only receiving messages and not transmitting) as opposed to active (both transmitting and receiving messages). This premise is very important in this method of secret key generation because as long as the channel between the two active users, for example Alice and Bob, is the same, then theoretically when they communicate in order to establish their secret key, the key will also be generated similarly. If an eavesdropper, for example Eve, is listening on a different channel then the same key would not be able to be generated since the signals traveling across Eve's channel will disperse differently than across Alice's and Bob's channel. In order to prove this theory multiple tests were conducted.

In each of these tests, in order to ensure that the channel between Alice and Bob is different from the channel between Alice and Eve and Bob and Eve, messages were sent back and forth between Alice and Bob and a sequence of 100 RSSI (Received Signal Strength Indicator) values were collected. At this time, Eve was listening to their communications and also gathering 100 raw RSSI values. All three sets of data were then quantized to a bit sequence of 1's and 0's based on their median value and compared to one another. In this comparison the similarities between the bit sequences would be observable. When comparing these sequences a similarity closer to 100% or 0% would imply that the channel between the users was the same and a similarity closer to 50% meant the bit sequence was random and that the channel for communications was not the same. Where any bits differed between Alice and Bob's sequence they could later be corrected with error correction coding. Multiple experiments would be conducted in order to observe this result.

With these preliminary results, this method for key generation could be explored and any potential challenges with this method could be addressed. This method was implemented in order to observe the correlation of the channels between Alice and Bob, Alice and Eve, and Bob and Eve. From this, it was shown that when Alice and Bob were communicating, their measurements exploited their channel similarities. When Eve was eavesdropping, her measurements exploited the differences in her channel compared to the channel between Alice and Bob. These results can be further explored as a method for securing wireless communications as more scenarios can be tested with Eve as an active user and with more nodes in the network.

1. Introduction

1.1: Wireless Communications

Wireless technology is the transmission of data via radio waves [1]. Various products and services in healthcare, defense, agriculture and other sectors heavily rely on wireless systems to operate. The wireless industry has benefited from the uptake and integration of technology and has grown in leaps and bounds since its inception in the 19th century. Figure 1 shows the results of a survey conducted by Cellular Telephone Industries Association (CTIA), an international non-profit body made up of industry stakeholders such as wireless carriers, providers and manufacturers of wireless data services and products [2]. The data represents 97.8% of all estimated wireless subscriber connections in 2014 [3].

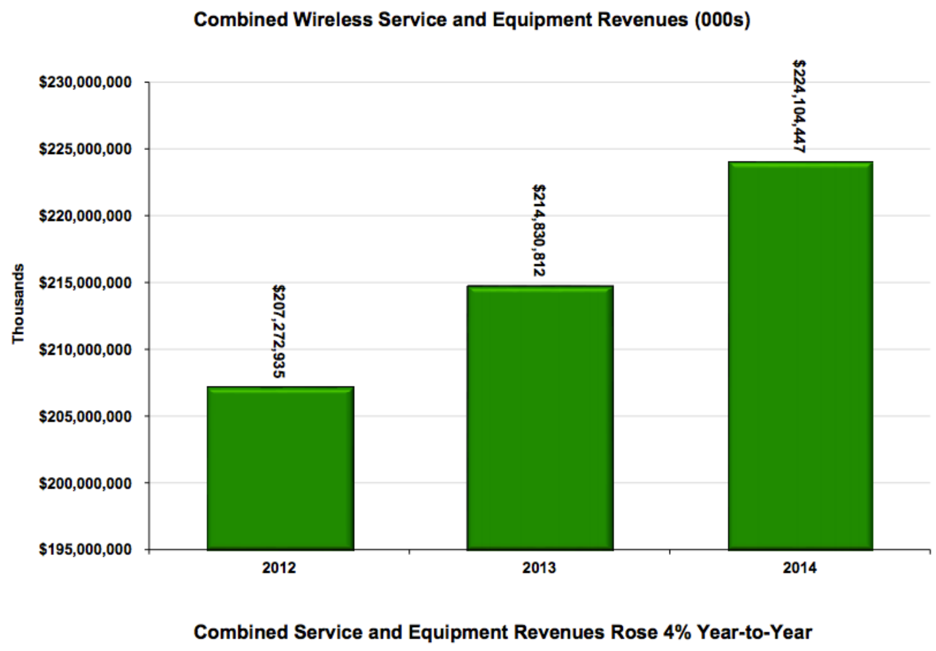


Figure 1- Graph of Combined Wireless Service and Equipment Revenues

Source: CTIA Survey 2014

The results reflect the incremental growth of the industry. More and more products are being developed that leverage advances in wireless technology to improve commerce and communication. These systems are creating wireless ecosystems that keep increasing and expanding the industry.

According to a survey conducted by Strategic Analytics in 2014, 451 million out of 690 million households worldwide make use of wireless technology to connect networks [4]. This number is roughly 65% of the population surveyed in the study. This study was performed in 27 countries and consisted of six global regions. If each of these households is assumed to have about three people, the data results in about 1.353 billion people using wireless technology. This information compared to the rest of the world's population, which was 7.2 Billion in 2014, shows that approximately 19% of the world's population makes use of wireless communications [5]. As the ubiquity of smart telecommunication devices increases, this percentage is sure to also increase. The scale at which wireless technologies are being employed is exponential. As a result, the security of these systems is of interest to both well and ill meaning individuals and groups and is a topic worthy of further exploration.

Security is at the heart of communication. Human civilizations have long since sought the ability to share messages whose meaning confounded any unwanted recipients. Decades of research and development has gone into making communication systems private and secure. With the advent of wireless communications, the need for security and privacy is even more evident and highly desirable. One difference between wired and wireless communication systems is the vulnerability to security breaches. Wired communication systems such as the Public Switch Telephone Network (PSTN) were much more difficult to tap because of the physicality of the network, making intrusion very visible and traceable. In the case of wireless communications, the medium of transmission is the air. This brings about more complexity in the support systems of the wireless communication and thus more avenues to breach the security of the system [6]. This mode of transmission also makes detection of attacks a more complex endeavor. Wireless communication security is thus very essential in modern communications.

At the heart of wireless communication are wireless devices. Wireless devices are devices that "communicate by transmitting electromagnetic signals through the air" (American Heritage Dictionary). Wireless devices can be put together in various configurations and implementations to make up a Wireless Sensor Network (WSN). WSNs have been described as collections of wireless devices joined wirelessly together for sensing metrics of the physical world and passing the gathered information to a main location [7]. Wireless Sensor Networks have applications in the military, agriculture, healthcare and many other fields. As research and development has been going on in the area of WSNs to provide more robust, efficient and reliable wireless devices there has also been development in the area of the "Internet of Things" (IoT).

The International Telecommunication Union defines the IoT as "[a] global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies" [8]. Academia and the industry are developing functionality and protocols that make the everyday things in the environment more connected. There has been ongoing development of connected cars and connected homes in the area of connected thermostats, alarm systems, home lighting systems and the like [9]. In these areas, more and more traditionally offline systems and devices are going online with a major characteristic of being wireless. This advent of IoT makes the topic of secure wireless communication even more pertinent. We explore the importance of security in the context.

1.2: Importance of Securing Wireless Communications

In communications there are two different types of networks that can be secured, wired and wireless networks. Securing a wireless network is much different than securing a wired network and also more difficult. The main difference between a wired and wireless network is that on a wired network the user's device has to be physically plugged in to an Ethernet cable, but on a wireless network no wires are necessary for connection. While there is more mobility on a wireless network, "[a]nyone can listen in, gather packets, and see what's [being sent] back and forth" across the network [18]. Therefore it can be seen that it is much more difficult to secure wireless networks where anyone can listen in on all communications. Wireless networking is becoming more and more popular with its desirable features such as mobility, and with this increase in demand, it is important to address the concern of wireless security.

A popular example of wireless technology used in households and corporations is Wi-Fi. In 2014, there were about 2.4 billion Wi-Fi enabled devices shipped globally and a cumulative number of 10 billion Wi-Fi enabled devices shipped by early 2015 [10]. Wi-Fi is a wireless technology which is used in the connection to a local area network and based on the IEEE 802.11 standard [11]. There are a number of ways to secure communications utilizing Wi-Fi. They are Wireless Encryption Protocol (WEP), Wi-Fi Protected Access (WPA) and Wi-Fi Protected Access 2 (WPA2). A core part of the WEP is the sharing of a secret key (a 64/128/256-bit long combination of hexadecimal characters) among the users in the network [12]. This protocol has been proven by researchers to be breakable in less than 60 seconds [13]. WPA was introduced to address the vulnerability of WEP. It introduced user authentication as well as key scrambling techniques but has also been proven to be unsecure and breakable in about 12-15 minutes [14]. WPA2 is, at the writing of this paper, the most secure way to encrypt Wi-Fi communications. It is still possible to break a WPA2 protected system but that requires greater computing power, thus making it more secure than the other two options [15]. WPA2 has been strongly recommended by the Information Assurance Directorate, a group within the National Security Agency, as the best way out of the three methods to secure Wi-Fi communications so far [16]. With the booming global uptake of wireless devices, it is of great import to understand the uptake of wireless network security and the kind of encryption being used.

The Wireless Geographic Logging Engine (WiGLE) is an on-line platform that gathers data on wireless networks around the world. As of November 2015, data from about 225 million wireless networks worldwide has been logged [17]. Of this number, 8.5% have no form of wireless encryption and 20% are using WEP or WPA encryption. The encryption of 19% of the networks was unknown. Table 1 reflects this information.

Table 1 - Breakdown of encryption utilized in a global wireless network sample population

Encryption Type	Number of Networks	Percentage (%)
WEP	27, 715, 606	12.33
WPA	20, 805, 221	9.26
WPA2	115, 097, 283	51.20
Unknown	42, 497, 472	18.90
None	19, 090, 113	8.49
Total	225, 205, 695	

https://wigle.net/stats#mainstats

The data reveals the state of security in the wireless communications domain and the dire need for methods that guarantee long term security that is based not on computational methods that can be declared unreliable at any given time, but on a more reliable method.

1.2.1: TJX Security Breach

With wireless security breaches becoming more and more common, the larger companies are beginning to be targeted. One company that has become a victim of wireless hacking is TJX. In the years 2005 and 2006 "TJX, the parent company of T.J. Maxx, Marshalls, and other [well known] retailers" [19] was wirelessly hacked and had over 45 million of their customers' credit and debit card numbers stolen over an 18-month period, making this the "largest customer data breach on record" [20]. One might wonder how the employees at TJX were not aware of this breach that was occurring for 18 months, however, in these days there are sophisticated ways of acquiring one's information wirelessly. This means that the person obtaining all of the information wouldn't have to step foot in one of the stores and could be as far as 45 miles away while hacking the company's information.

TJX was known to have secured its wireless network using what's known as WEP, which is one of the weakest forms of security for wireless local area networks (LANs). The wireless hackers used a technique called wardriving to decrypt the WEP protocol and obtain access into TJX's system, enabling them to set up their own account while their software was able to acquire "transaction data, including credit card numbers, into approximately 100 large files" [21]. Using this technique, the attackers were able to find a "vulnerable store location while staking out a strip mall or shopping center from their car using a laptop, a telescope antenna, and an 802.11 wireless LAN adapter" [21]. All the attackers had to do was drive around and point their antenna at different stores, and the antenna would detect the wireless access points from miles away and see how they were configured. According to security researchers, "once the attacker is connected into the wireless network, they can sniff traffic to see what data's going where" [19]. This is how an attacker knows where all of the information is going and where to concentrate their efforts. With all of this information obtained, it is unknown where it might go next, as this information can be bought and sold across the world.

Wardriving relies on the fact that the network has weak security. According to security researchers, "'when a company puts in wireless, they don't put it in securely ... [and] they forget that wireless is yet another way in" [19]. The TJX store was "using an outdated protocol that's notorious for allowing small amounts of data to leak from data packets flowing across a wireless network" [19]. The inadequacies involved with WEP have been known for years and WEP has been known to forfeit over the encryption key when attacked. There are many unsecure networks and this example shows why it is important to secure wireless networks and communications to prevent leaking private information that could potentially be damaging.

1.2.2: Businesses Hacked by Wardriving Wi-Fi Networks

In 2011 "three men were indicted...by a federal grand jury for hacking at least 13 Seattle-area businesses' wireless networks to steal sensitive information" [22]. The attackers were able to steal "credit card numbers and payroll information via the businesses' wireless networks, enabling them to steal more than $750,000 in cash and computer equipment, among other items" [22]. In obtaining this information, the attackers used the technique known as wardriving where they were able to sit in their car along with antennas and other network tools to pick up on wireless networks to see if they were vulnerable. Again, as in the TJX hack, the attackers "target[ed] networks secured using Wired Equivalent Privacy (WEP), a 12-year old, outdated, and unsecure standard, which is still used by many Wi-Fi routers" [22]. Once the wireless network was able to be accessed, software to open ports on the server or host was run as well as software to recover passwords. The attackers could then gain access into the businesses' private information and records. With this, more than 45 million credit card numbers were obtained. One of the businesses' that was affected by this breach, Concur Technologies, had approximately 1,017 of their employees' names, addresses, dates of birth, and social security numbers stolen.

With two major wireless security breaches taking advantage of the outdated and unsecure WEP network security, the question as to why companies are still using this method to secure their wireless networks arises. Security researchers advise companies, especially big companies, to use "more sensible, hardened encryption, if [they're] going to have wireless communications" [22]. This leaves room for advancement in this area, and if a reliable and secure approach can be established to encrypt wireless communications, companies would not have to fear that valuable information or their employees' identities would be stolen and instead can feel reassured that their wireless security will keep out attackers.

1.3: Two Types of Security Approaches

With many examples of companies' Wi-Fi networks being hacked, it is important to find ways of securing wireless communications to ensure that this does not happen. As previously mentioned, the companies that were targeted used WEP to secure their Wi-Fi networks, however, this security approach was not impenetrable. WEP uses a security algorithm to generate keys for encryption and decryption of messages. The ways in which these keys are determined in wireless communications plays a major role in how vulnerable they are to attackers. In wireless security there are two main types of techniques to generate these keys, computational security and information theoretic security, as each will be explained in more depth in the following sections.

1.3.1: Computational Security

Of the different types of securities, computational security uses concepts from cryptography to securely encrypt wireless devices [23]. Computational security involves developing complex mathematical algorithms where, ideally, it would take an attacker an unreasonable amount of time to decrypt the algorithm and maintain access to the communications between devices. While this method is not meant to indefinitely encrypt a device, the cryptographic method "incorporates an integer security parameter" which helps to ensure secure communications [24]. This security parameter is a key, and when an attacker is able to learn this key is when key recovery has taken place, and the attacker is then able to eavesdrop on all communications between the devices. This poses a problem for secure devices, and to ensure that this doesn't happen, encryption should be secure against key recovery [25]. As an attempt to ensure this security between wireless devices, there are different types of key encryption techniques that are utilized in computational security.

Two types of key encryption techniques are symmetric key-encryption and public-key (or asymmetric key) encryption. Symmetric key encryption is a method where the sender and the receiver share the same key or where their keys are slightly different, but mathematically related. The same key is used for both encrypting and decrypting messages, however the disadvantage of this method is that it is "difficult to securely establish a secret key between the two communicating devices when there does not already exist a secure channel for communications" [26]. Another type of key encryption technique is public-key (asymmetric key) encryption. In this method, the sender and receiver both develop two different keys, a public key and private key that are mathematically related. The public key is used to encrypt messages while the private key is used to decrypt messages. These two keys are generated so that, even though they are mathematically related, the private key cannot be determined from the public key [27].

While these computational algorithms and techniques may seem secure against attackers, they are not full-proof. Concerns arise with this encryption method because an attacker could decrypt the messages by solving the complex algorithm used to generate the keys. Once the attacker has solved this algorithm, it can eavesdrop on all communications.

One major concern with the stability of computational security is the viability of quantum computing. In recent times, there has been much progress in the field of quantum computing. As quantum systems are made more practical and ubiquitous, they will strongly test the current security systems which rely heavily on unsolvable algorithms. Current research shows that for certain problems, quantum computers are much faster at finding solutions than personal computers [28]. The fact that quantum computers, which have recently just come out, can match and at times surpass, the computing power of personal computers, which have had years of architectural refinement and performance optimization, raises concerns about the standing power of computationally secure systems in the coming years [28]. It can therefore be concluded that in a few years, as more research is performed into quantum systems, our current systems may not stand strong against adversaries with quantum computing power. It is in this light that information theoretic security lends itself as a more secure approach to security.

1.3.2: Information Theoretic Security

Information theoretic security is a form of securing a system using tools from information theory. The mathematician, Claude Shannon first introduced this approach in 1949. In computational security, a large portion of the security is based on an attacker's inability to break down an algorithm due to its mathematical intractableness or an attacker's inability to possess enough computing power to solve the complex mathematical algorithm. Information security on the other hand, ensures security of the system regardless of the attackers computing power and makes no assumptions on the intractableness of a mathematical algorithm [29].

An example of an information theoretically secure system is the "One-Time" pad [30]. In the One-Time pad, a random key, which is the length of the message that will be sent (plaintext), is chosen. The key is then used in a modular addition (xor) with the plaintext. The result is the message transmitted over the channel (cipher text). Both parties, Alice and Bob, for example, have access to the random key used for the encryption and can each decipher one other's message provided the shared key is used to secure the message. The adversary, for example Eve, has no information before or after receiving the cipher text. The One-Time pad has been shown to provide perfect security provided no part of the key is used to encrypt a message again [30]. In such a system, an attacker's access to unlimited computing power is not enough to break the system as he/she will still not have access to the key information. This approach has been used for diplomatic communications as it is a highly secure method [30].

An essential component of a computationally secure system is the encryption key. Knowledge of this key by the attacker compromises the security of the system as such key generation and protection is of major concern. In information theoretic security, a key is also used to secure the system. The key however, is generated not through computational processes. One method of key generation is to observe the environment in which the communication between Alice and Bob is taking place and to leverage the changes of the communication signals introduced by the environment. The environment, in this sense, is referred to as the channel. A common phenomenon within wireless communications is changing signal strength from the transmitter to the receiver. This can cause signal drops during cell phone calls as well as low Wi-Fi signal strength. This phenomenon is due to the multiple paths a signal takes before reaching the receiver and can be due to interaction of moving components such as cars and human beings within the channel.

Estimating the signal changes within the channel provides rich information that can be used to generate a secret key for securing the communications. As the channel between the communicating parties change, new secret keys can also be generated to encrypt messages. This ensures that a system with very low vulnerability to attacks can exploit key reuse. This approach for key generation is based solely on the information in the channel and not from computational processes thus devoid of the inherent weaknesses outlined in computational security methods.

1.4: Our Approach

In our approach, the randomness of the channel gain between the communicating parties is exploited to generate a secret key. The randomness of the channel is observed by two users, Alice and Bob, where, through communications, will be able to accurately estimate the channel in which they are communicating. From this estimation, with high probability, the same key will be generated by both parties using tools from coding theory [31]. This key would then be used to decrypt and encrypt messages being sent back and forth. Since Alice and Bob are communicating through relatively the same channel there will be similarities that each party will be able to observe regarding the channel. An eavesdropper, Eve, will be trying to listen in on the communications between Alice and Bob, however, Eve will be listening on a different channel in some other location. This figure can be seen below with the orientation of each party.

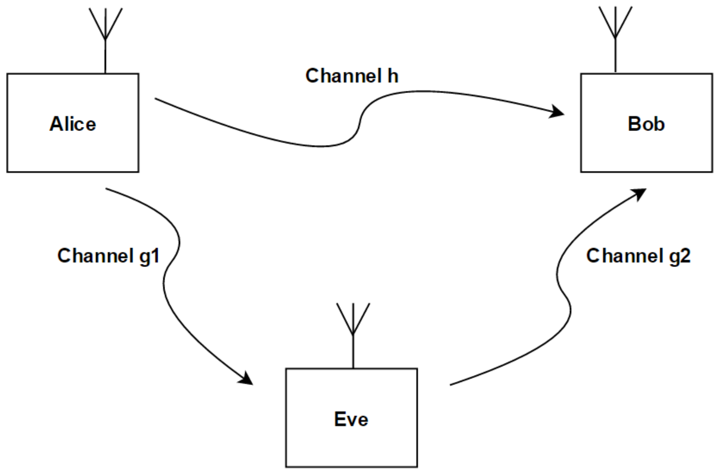


Figure 2 - Orientation of two communicating parties, Alice and Bob, and eavesdropper, Eve

From this figure it can be seen that Eve will experience a different channel. This is expressed in Equations 1-3 where h, g_1, and g_2 correspond to the channels labelled in Figure 2, s is the signal being sent, n is the noise of the channel, and y is the signal that is seen at the receiver.

$$y = h * s + n \tag{1}$$
$$y = g_1 * s + n \tag{2}$$
$$y = g_2 * s + n \tag{3}$$

Due to the fact that the key was generated based on observations of a specific channel, this same observation will not be able to be replicated, and therefore the same key to decrypt the messages will not be able to be generated. Even with slight movements from either of the communicating parties, Alice or Bob, or with movements from objects between them, the channel, in response, will change. With this change the path of the signal as well as any scattering, noise and channel gains will also change and hence, the key will change. Alice and Bob will be constantly sending messages back and forth and from this constant communication they will be able to estimate the new channel and with as high a probability that the channel was estimated with, they will be able to regenerate a key [31].

With the channel between Alice and Bob changing and with Eve on a different channel, this will make it nearly impossible for Eve to generate the same key as Alice and Bob to eavesdrop on their communications.

The purpose of this project is to create a prototype testbed to implement the above mentioned idea. The remainder of the paper is organized as follows. In chapter 2, we present different transceiver options to use in this project and the criteria used to choose the final transceiver. In chapter 3, we discuss the sensor setup and software implementation. In chapter 4, we present the results that were gathered. Finally, chapter 5 summarizes the project and discusses possible future work and recommendations.

2. Transceiver Options

In this chapter, we provide details on how we choose the platform for implementation and how we select tools for the chosen platform.

2.1: Received Signal Strength Indicator (RSSI)

Received signal strength indicator (RSSI) is a very important measurement in communications as it is the “measurement of the power present in a received radio signal” thus measuring the quality of the received signal [32] [33]. The received signal strength is measured at the receiving device as this measurement is usually taken at the intermediate frequency stage [32]. RSSI is oftentimes available to users via wireless networking cards or some other wireless network monitoring tool such as the Wireshark, Kismet, or Inssider [32]. With this information about the signal strength received at the end user, for example at a smartphone, it can be determined if the signal strength is strong (a high RSSI value) or if the signal strength is weak (a small RSSI value). These RSSI measurements are often recorded in milliwatts (mW) or decibel-milliwatts (dBm). When measuring the received signal strength in dBm, it is appropriate to conclude that the closer to 0 dBm the received signal strength is, the better the signal [33]. Depending on this information, wireless devices can determine if the energy is below a certain threshold and if the device is “clear to send” [32]. Once the device has deemed that it’s clear to send, the wireless device will then send its packet across the channel.

Once the packet is transmitted, there are many different channels that the signal can be sent through, for example air or water, and depending on which medium the signal has been sent through, the measurement for RSSI can be greatly affected. One scenario where the RSSI may be affected is when there is multipath propagation where there are multiple paths from the transmitter to the receiver as well as obstacles in the way such as walls, buildings, cars, etc. that the signal can bounce off of, changing the signal's path and direction. From the RSSI measurement the distance between two communicating devices can be approximated, and the more multipath propagation that the signal endures, the longer the signal has to travel to get to the end user, therefore resulting in a lower RSSI value [34]. The information determined from the RSSI value is very valuable in wireless communications systems, and was also an important factor in this project.

For this project the RSSI value would be used as the estimate of the channel gain, as this measurement is directly proportional to the channel gain. The RSSI measurement would reflect the quality of the channel, and as long as the two users received the same measurement, this meant that their channels were similar. The similarity in their channel gain is very important for successful key generation and because of this, ensuring that the transceiver could sense RSSI values was an important factor to consider when choosing a sensor.

2.2: Platform Selection

For this project different platforms were researched to be used as the three transceivers, Alice, Bob, and Eve. There were many different criteria that these platforms had to meet in order to be considered, which will be discussed in more detail in the following sections. After extensive research, two different types of platforms were decided upon, Software-Defined Radios and sensors. These platforms were further narrowed down to determine which of the two would be best suited for communications in this project.

2.2.1: Sensor Selection

There are various sensors that could be selected to implement this project. The criteria for this selection becomes stringent when considering sensors to create an indoor wireless network. A few factors that were considered in the evaluation of sensors were as follows, in no particular order of preference:

- Physical unit (complete module or incomplete unit)
- Operating frequency and wireless standard
- Ability to measure RSSI
- Presence of microcontroller (MCU)
- Operating system and programming language of MCU
- Development environment
- Online support for MCU and programming language
- Sensor costs

The physical unit of the sensor played an important role in this choice. A sensor consisting of an antenna, microcontroller, serial interfaces (USB) and any other wireless network components all on a complete module-based platform was highly desirable. Such a modular sensor would remove the errors that could arise in the experiment from incompatibility of components.

The operating frequency and wireless standard of the sensor was also of great importance. As the goal of the project was to secure wireless communications, the chosen sensor had to operate in the frequency bands used in common wireless sensor networks, ~2.4GHz and ~5GHz. Operating in this frequency range would make the results gathered reflective of practical everyday situations. The ability of the sensor to measure the RSSI was also a key factor in choosing an appropriate sensor. As discussed earlier, RSSI is representative of the power of the received signal. The accessibility of this value was greatly considered when deciding on a final sensor, as this value will be important in subsequent calculations of channel behavior estimation. It was very essential that the sensor measured RSSI for the success of the project.

The presence of a microcontroller unit (MCU) on the sensor platform was also considered. The type of MCU dictated the operating system (OS) of the sensor. The OS in turn, determined the kind of programming language and paradigm that was used. These specifications were used to determine the amount of online support for the sensor software, development environment, and the programming language, which would help to become familiar with them if the team had limited knowledge of these platforms.

Another main factor that was considered was the cost of the sensor. As in the set-up described in Figure 1.1, three sensors were needed for this project. The project team had a budget of $250.00, and because of this limited budget sensor costs had to be taken into account when deciding on a sensor.

Research into previous projects in wireless networks was conducted to determine suitable sensors for this project. The most common sensors found to be used by researchers in this field were wireless cards with antennas such 802.11n Intel WiFi Link 5300 Wireless card [35] as well as standalone complete wireless modules such as MICAz [36] Tmote Sky built off TelosB platform [37] [38] and iMote2 [39].

Tables 2 and 3 show various sensors that were compared and their respective specifications for the criteria considered.

Table 2 - Comparison of sensors and their characteristics

SENSOR/ CRITERIA	Complete Module	Operating Frequency (GHz)	Measuring RSSI	Wireless Standard
MAX2410	No	0.8-2.4	No	Cordless & PCS standards
MAX2828	No	4.9-5.875	Yes	802.11a
MAX2829	No	2.4-2.5 & 4.9-5.875	Yes	802.11a/b/g
MICAz	Yes	2.4-2.48	Yes	802.15.4
TMote Sky (CM3000)	Yes	2.4-2.483	Yes	802.15.4
iMote2	Yes	2.4-2.483	YEs	802.15.4

Table 3 - Continuation of table 2

SENSOR/ CRITERIA	Microcontroller	Operating System	Programming Language	Development Environment	Cost per Sensor
MAX2410	N/A	N/A	N/A	N/A	Free sample
MAX2828	N/A	N/A	MAX2828/29 Control Software	N/A	Free sample
MAX2829	N/A	N/A	MAX2828/29 Control Software	N/A	Free sample
MICAz	Atmel ATmega 128L	MoteWorks, based on TinyOS	nesC	Eclipse (Yeti2 plug-in), CCS, Linux	$95-$150
TMote Sky (CM3000)	MSP430F1611	TinyOS Support	nesC	CCS	$90.22
iMote2	Intel PXA2771 XScale Processor	TinyOS, Linux and SOS	nesC/C	Eclipse (Yeti2 plug-in), CCS, Linux	$990 (bundle)

After carefully comparing each of the different sensors, the search was narrowed down further based on the importance of the criteria for this project. One important feature of the sensor that was necessary for this project was that the sensor came as a complete module; having a microcontroller connected to the transceiver and mounted on an evaluation board so that it could be easily programmable and the information could be easily processed. Another aspect that was of importance was that the sensor could measure the receiver signal strength (RSS) and that this measurement could be easily accessible through a receiver signal strength indicator (RSSI). From the six sensors originally researched, only three met both of these criteria, the TMote Sky (CM3000), MICAz, and iMote2. These three sensors were then researched more in depth in order to decide on one of these sensors to be used for testing.

With these three sensors in mind, they were then narrowed down further based on the microcontrollers that were implemented. In looking at the microcontrollers, the programming language as well as the development environment also had to be taken into consideration. The microprocessor, programming language, and development environment all had to have sufficient support for both interfacing with the sensor and for the language used in programming the microprocessor. With this, the learning curve couldn't be too difficult so that, with limited background and knowledge on the programming language and development environment, the sensor could still be properly programmed and configured for the purpose of this project.

The final criteria that had to be considered were the prices of each of the sensors. With a limited budget of $250.00 and the need for three sensors to properly implement this project, the price of the sensor was very important. The sensor had to be cheap enough so that three sensors could be purchased, but at the same time still fit the rest of the criteria that was previously mentioned. In researching the prices of the sensors, the cost of one sensor, the iMote2, could not be found and because of this, further inquiries were made to the developers. From these inquiries it was determined that this sensor was not currently being manufactured, and a new sensor that would replace the iMote2 was still in the research and development stages. Based on this news, the iMote2 was eliminated from the search.

With these final requirements in mind one of the final two sensors, the MICAz and TMote Sky (CM35000), were chosen. Both modules were similar in regards to all of the criteria, except one, price. The MICAz was more expensive than the TMote Sky (CM35000), and with the limited budget, only two MICAz sensors would be able to be purchased when three were needed for this project. The TMote Sky (CM35000) sensors weren't as expensive and because of this, three sensors could be purchased for this project. After careful consideration, the TMote Sky (CM3000) was chosen for testing secure wireless communications between two parties with a third party eavesdropping on this communication. The TMote Sky (CM3000) can be seen below with a USB adapter, the USB1000, connected to the Erni adapter.

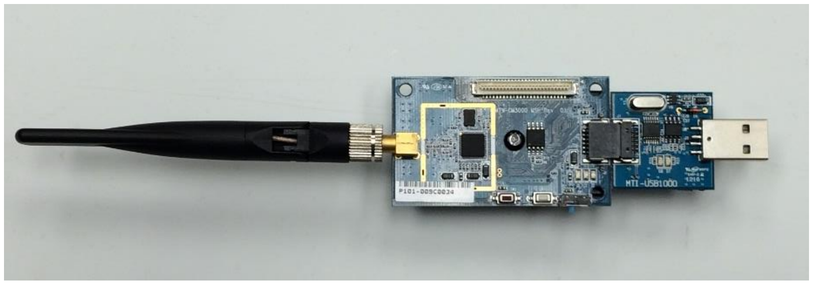


Figure 3 - TMote Sky (CM3000) sensor

This sensor met all of the needs of this project and was further tested against the software-defined radio to determine which would be used in the final implementation.

2.2.2: Software Defined Radio Selection

As well as considering different sensors to utilize in this project, software-defined radios were also considered as an option to use as the transceiver. According to the SDR Forum in collaboration with the Institute of Electrical and Electronic Engineers (IEEE), a clear definition for software-defined radios have been established. Software-defined radios are "radio[s] in which some or all of the physical layer functions are software defined" [43]. Software-defined radios have the capability to transmit and receive messages over different frequencies depending on the daughter card that is used in the radio and is very desirable for use in high-frequency applications.

Due to the fact that software-defined radios are very costly, only two radios, the USRP 2 and USRP N210 were considered for use in this project because they were readily available. Both of these software-defined radios were created by Ettus Research and are also compatible with different software and software development tool kits such as MATLAB, Simulink, GNU Radio, and LabVIEW. This software can be used with different programming languages such C, C++, Java, etc. to aid in the development of applications to implement in conjunction with the software-defined radios. Some of the applications where software-defined radios are used is in the development of adaptive radios and cognitive radios, and although SDRs are not necessary in these applications, SDRs "provide these types of radios with the flexibility necessary for them to achieve their full potential" [43].

With the two SDR's available, it then had to be decided which one would be used in this project. Both the USRP 2 and the USRP N210 were compatible with all of the above mentioned software and software development tool kits. One difference, however, between the USRPs was that the USRP 2 utilized a graphic SD card, where each had to be individually burned and remain plugged into each of the USRP 2s while in use. This additional step was not necessary for the USRP N210. While this was the only main difference between the USRPs in reference to technical aspects, another difference was the availability of each of the USRPs. The USRP N210, the newer model of the two, was being widely used throughout the school. This meant that classes that ran during the course of this project would be using them the vast majority of the time, limiting their availability for this project. The USRP 2s, on the other hand, were not currently being used and thus were readily available. With these considerations, more research into the USRP 2 was conducted, both the files and software were found in order to flash each of the graphic SD cards for the USRP 2, and a final decision was made. In the end, time was the most important factor, and therefore it was decided to use the USRP 2. The USRP 2 software-defined radio can be seen in Figure 4.

Figure 4 - Software-defined radio (USRP 2)

The USRP 2 was then further compared with the TelosB sensor to determine which platform would be the best fit for this project.

2.2.3: Final Platform Selection

The TMote Sky and USRP 2 were both capable sensors for the purposes of this project. A comparison of the hardware and software characteristics of both sensors against the goals of the project was explored.

One major goal in this project was channel estimation. The ability to move the sensors was a very essential characteristic that the chosen sensor had to possess. The sensor would be placed in different physical locations and scenarios and the measurements of transmitted and received signals taken. The TMote Sky proved to be more mobile in nature and could also be deployed in the field with little to no dependence on a supporting computer setup.

The ability to access and easily measure the RSSI value of the transmitted signal was another critical component of the sensor. It was more challenging for the team to access the RSSI value from the USRP2 and the supporting MATLAB software. The TMote Sky on the other hand provided easy access to the RSSI value via software commands.

The TMote Sky sensor was programmed with the nesC language, which was a variant of the C language. Applications on the TelosB can have a Java layer. This allows access to the extensive libraries and support of the Java language. There is also a good online support system for TelosB sensors, TinyOS and the nesC language [44]. To use the USRP2 for essential measurements such as RSSI, a possible approach would be to program the USRP 2 in C++ via the GNU Radio method. Support for this approach was available via GNU Radio [44]. To learn and efficiently utilize this approach would take more time compared to the TelosB sensor's technologies as the project team was already well versed in the C language and Java. The TMote Sky (TelosB) was chosen as the sensor to implement the project as it had more desired hardware and software characteristics in comparison to the USRP2.

2.3: Implementation Platform

In this section, we discuss the implementation platform of the chosen sensor.

2.3.1: Operating System

The TMote Sky uses the TinyOS operating system (OS). TinyOS, is an open source, component based OS targeted for low-power sensor motes typically used in sensor networks and ubiquitous computing among other applications [40]. TinyOS is written in the nesC language, a dialect of the popular C programming language. Applications in TinyOS can also be written in C [40]. TinyOS also allows for interfacing with Java and shell script programs [40]. One key benefit of TinyOS, is the abstraction layer it provides. It is able to represent device hardware as software abstractions making software development in TinyOS simple. An example is with a flash storage chip. TinyOS is able to represent this device as a circular log software abstraction that can be manipulated via software. In addition to being used for low-power sensor motes, TinyOS is also useful for microcontroller based devices with limited resources such as the MSP430-based sensors which run a small amount of memory [40].

Advantages of TinyOS

TinyOS has other strong advantages as an OS for sensors. Currently, the OS can be installed on Linux, Mac OS X and Windows computers. It is also possible to use virtual machines to run the OS. The latter approach is much simpler than installing the OS onto a computer. TinyOS is able to run on various sensors such as the TelosB, IMote 2, Micaz, IRIS, Mica2, the Shimmer family, Epic, Mulle, tinynode and Span family of sensors [40]. TinyOS also supports various microcontrollers including the Texas Instruments (TI) MSP430 family, Atmel's Atmega128, Atmega128L, Atmega1281, and the Intel px27ax [40]. The OS also has support for radio chips including TI/ChipCon CC1000 and CC2420, Infineon TDA5250, the Atmel RF212 and RF230, and the Semtech XE1205 radio chips. User groups have also included support for the TI/ChipCon CC1100 and CC2500. TinyOS also provides support for two NOR flash chips, the Atmel AT45DB and STMicroelectronics STM25P chip. As such, there is a wide range of support for running the OS on the different sensors and radio chips [40]. The OS also has strong support for the CC2420, a popular 802.15.4/ZigBee radio chip. This is also the chip in the TMote Sky sensor used for this project. The OS has also been in use for over five years and has a strong and efficient code base [40]. It is a popular OS among researchers and commercial users. Some examples of commercial users are Motorola, Intel and Crossbow [40]. TinyOS is also a very good OS for networking and radio communications. By possessing low-power link layers, the OS supports low duty cycle operations. The OS accomplishes this by turning on the radio for certain periods of time to check if there is a packet to be received. As such, the radio appears to be always on and can still have sub-1% duty cycle [40]. The tradeoff with this method is that there is latency in communication applications.

Disadvantages of TinyOS

There are significantly, two major weaknesses of TinyOS: the learning curve of the programming model and the difficulty in writing computationally-intensive applications. The Application Programming Interfaces (APIs) of TinyOS are split-phase or non-blocking. This is because, TinyOS runs on devices with small amounts of RAM. In a non-blocking procedure, threads are allowed to access shared resources without blocking the other threads. Also, the failure of a thread does not cause the failure of another thread [41]. An example is in the sending of a message in TinyOS. A send function is used in the sending of the message. When the function is called, it returns immediately before the message is sent. After some time, TinyOS calls sendDone, a callback on the procedure, telling it that the message has been sent. In order to have this non-blocking behavior, TinyOS does not expect portions of code to run for long periods of time uninterrupted. If this happens, other events such as sendDone cannot access resources. This affects the performance of the OS and may lead to packet dropping and missed timeouts [40]. As most novice programmers are not familiar with this event-driven, non-blocking procedure, learning and applying the concept can take some time.

Computationally intensive procedures also have to be structured differently as the structure of TinyOS does not tolerate pieces of code running for long uninterrupted periods of time. To be able to have such procedures, the program has to be broken into smaller pieces that are executed one at a time. For example, to implement a nested for loop which runs over NxN array, you can have N separate computations that each run over an array of length N [40]. A solution to the difficultly in understanding the programming model of the TinyOS is the usage of the large amount of tutorials and supporting documentation for the TinyOS. This can help novice programmers come up to speed with the OS and learn more efficient ways of writing applications. There is also a library in current versions of TinyOS that supports threaded applications. It enables programmers write threaded applications on top of the OS. The ability to have a blocking API and longer loops are part of the functionality the library offers. Applications built with this library can also be developed in the C language. As this language is more popular than nesC, it helps reduce the learning curve in building TinyOS applications.

2.3.2: Programming APIs in TinyOS

In TinyOS, operations which can take a long time are split-phase. This means that they have completion call-backs. The first phase is the command which starts the operation and the second is the callback which signals the completion of the procedure. In most APIs, the function is bound at compile time and the callback is passed as a function pointer. In nesC both functions are bound at compile-time [40]. nesC is made up of nesC interfaces. These interfaces connect software components that make up the application to be run. For example, an application has component A and component B. There are calls from component A to B and from component B to A. These calls are specified by the interface. Some examples of interfaces are send/sendDone, for sending packets, start/fired, for starting timers, read/readDone, for sampling sensors and write/writeDone for storage [40]. In these examples, the functions with "Done" are the callbacks and the functions without "Done" are the starting functions.

2.3.3: nesC Language

The nesC language is a dialect of the C language. It has the basic statements of the C language including for loops, variable declaration and assignments. nesC uses a component based programming module where the application is split into software components. These components join code with state and are only instantiated at compile time [40]. The language also has interfaces. These interfaces specify roles and services within the application. Interfaces are bi-directional in nature and allow for components to interact with each other. nesC also utilizes a concurrency based on that of TinyOS. The language is able to differentiate between code sections that cannot be acted on by interrupts and those that are not runnable in the context of interrupts [40].

2.3.4: Structure of a nesC Application

A nesC application is made up of a one or more components that make up its application executable [45]. There are two kinds of components: configurations and modules. Both of the components provide and use interfaces [46]. Modules implement the interface(s) that make up the application. Modules contain the code that the application executes. Configurations "wire" together components by connecting the interfaces that the components use to those that are provided by others [42]. Code specified in the components are private and can only be accessed in those components. In order to name or access code in components directly, interfaces are used. [46].

2.3.5: nesC vs C Language

There are two main reasons why nesC is used in programming TinyOS than the C language: the linguistic support that nesC provides and the strong code optimization [40]. An example of linguistic support is in the programming of events and tasks. In these areas, C uses macros which are susceptible to bugs. nesC on the other hand ensures the compiler checks the usage of tasks and events. This results in less bugs.

Using nesC also allows the production of C code which can be better optimized by the compiler as compared to just using the C language. Function inlining is an example of this. Usage of inlining functions result in faster execution time and reduction in the code size of the application [42]. Some compilers of the C language such as gcc, only inline functions if they were already defined [40]. nesC has strong inlining characteristics and inlines smaller functions [42]. As application development in TinyOS makes use of small functions, this characteristic of nesC is very desirable.

3. Implementation

In this chapter, we discuss the details of our implementation.

3.1: TinyOS Installation

Before the sensors could communicate with each other, TinyOS needed to be setup on the computers. After much research on the TelosB (CM3000) sensor, it was decided that the sensor would work best running on a Linux-based operating system. Therefore, Ubuntu, an open source, Debian-based Linux operating system that is free for anyone to download [47], was installed on all of the computers that were used in this project. Once Ubuntu was installed, TinyOS was then able to be installed so the sensors could be used.

Using the Advanticsys online resources, the *TinyOS Installation Guide* was used to aid in the installation of TinyOS. This installation guide can be found in [48]. All of these steps were carried out in the terminal window. First, the command `sudo -s` was run in order to login as the root user, as this was needed to make the necessary changes required during installation. Next, under the "Ubuntu Linux Environment" section in the *TinyOS Installation Guide*, steps one through four were carried out. After step four was completed, a YouTube tutorial that was found online was used to finish the TinyOS installation. This video can be found in [49].The rest of the installation could be completed using the YouTube tutorial as a guide. These steps were well documented in a file with tips to make the installation run smoothly and to help with problems that could be encountered along the way. This document can be found in the appendix labeled "Steps for TinyOS Installation on Ubuntu". Once the installation of TinyOS was complete, the sensors were then able to communicate with the computers and with each other.

3.2: Sensor Setup

In order to get accurate measurements, the sensor setup was very important. Overall, there were three sensors, Alice, Bob, and Eve. Alice and Bob were the two main communicating sensors and Eve was the eavesdropper, trying to listen in on communications between Alice and Bob. There were three different setups that were used for testing.

One scenario just looked at communications between Alice and Bob where both sensors were stationary and each were plugged into the desktop computers. This sensor configuration can be seen in Figure 5.

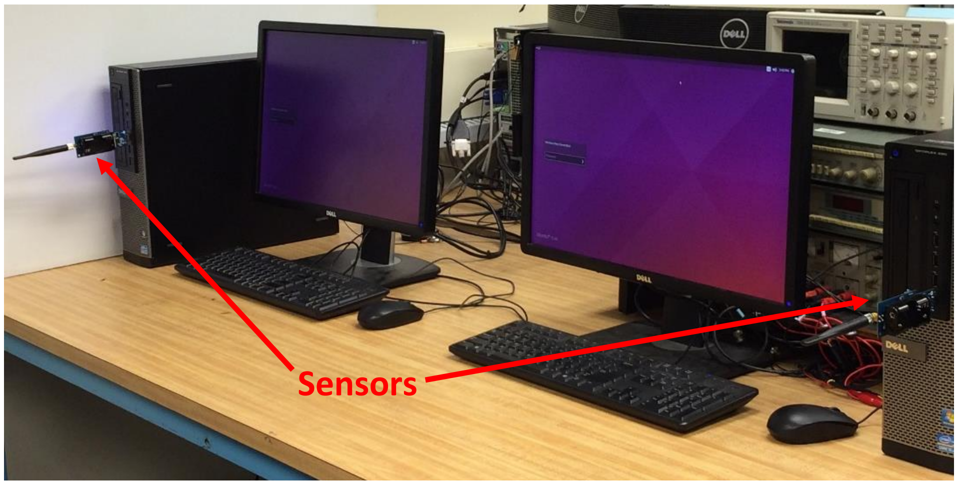


Figure 5 - Sensor setup for testing between Alice and Bob

The second configuration was where Bob was stationary and Alice was moving. To accomplish this, Alice was connected to a laptop and the distance from Alice to Bob was continuously increased as Alice moved further away from Bob. In doing this, the effects that distance had on the communications and key generation could be observed. This sensor setup can be seen in Figure 6 where Alice is connected to a laptop and Bob is connected to the desktop computer.

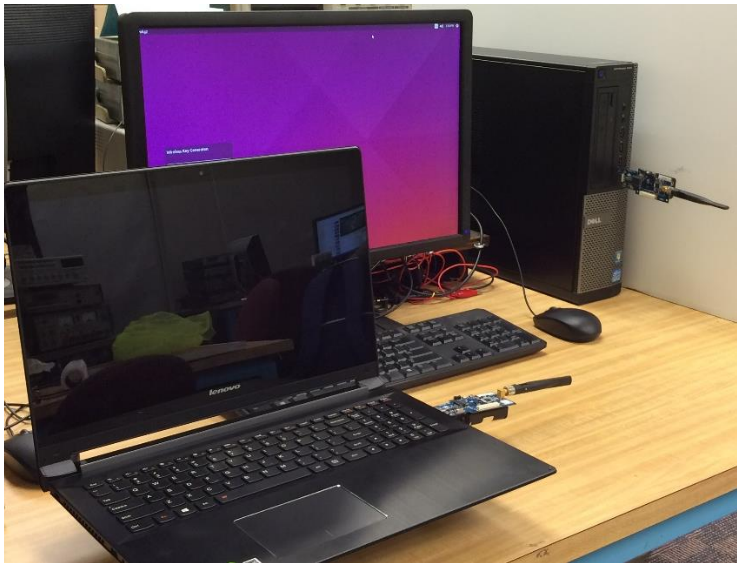


Figure 6 - Sensor setup for Alice (laptop) moving and Bob fixed

A final sensor configuration was considered where Alice and Bob remained at fixed positions (in this case connected to the desktop computers) and where Eve tried to listen in on their communications while connected to a laptop. In one setup Eve was located at a fixed position and in another setup Eve started out close to Alice and Bob and then moved further away. These different sensor configurations can be seen in Figure 7.

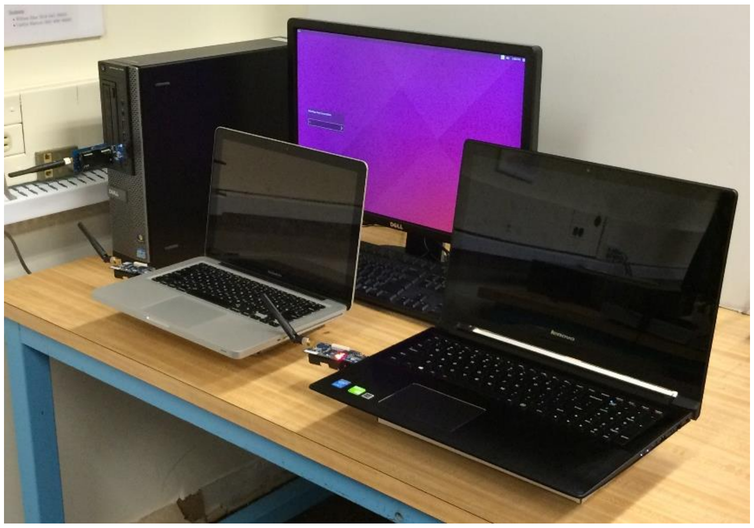


Figure 7 - Sensor setup for Alice and Bob fixed while Eve (laptop) is moving and trying to listen in on communications

With these different configurations it can be seen that each sensor had a different channel for communicating every time the environment or sensor location was altered. With the sensors located at different positions multiple tests could be run in order to observe the similarities and differences in their channels and in the key that was generated.

3.3: Channel Estimation

To generate the common key between Alice and Bob, they need to obtain an accurate estimate of the channel gain between them. To achieve this, Alice and Bob will take turns to send constant signals while the other terminal will obtain RSS, which will be used as an estimate of the channel gain.

The Tmote Sky used in this project has a CC2420 chip. This chip has a built in Received Signal Strength Indicator which it is provides via an 8 bit register called the RSSI.RSSI_VAL register [50]. The RSSI value is calculated by averaging over 128us (8 symbol periods) in accordance with the IEEE specifications for Low Rate Wireless Personal Area Networks (LR-WPANs) [51]. The register calculates the raw RSSI value for each symbol received. According to the CC2420 data sheet, the RSSI value in dBm can be referred to as the power of the received signal at the Radio Frequency (RF) pins by Equation (4):

$$P = RSSI_{val} + RSSI_{offset} \; [dBm], \qquad (4)$$

where P is the power in dBm of the received signal at the Radio Frequency (RF) pins and $RSSI_{offset}$ is the offset which the data sheet states to be -45 [50].

For example, if the RSSI value obtained from the register was -30, the RF power of the signal was -30–45 = -75dBm. This result after accounting for the offset is the RSSI value in dBm that is of interest to Bob.

Two applications were written in nesC for sending and receiving messages between Alice and Bob. A supporting Java file was used to retrieve the values of the RSSI in dBm from the sensor, observe the values in a Linux terminal, and log the values to a file for post-processing. The nesC and Java codes are the same at Alice and Bob's end. Each sensor is connected to a computer for the entire duration of the communication. The nesC portion of the project was built on top of tutorial code provided in the installed TinyOS package. This tutorial is called BlinkToRadio. It demonstrates back and forth communication between two sensors within range of each other. In this tutorial one sensor increments a count variable in software, creates a radio message containing the variable value and transmits the message over the air (radio). When the other radio receives the message, it obtains the payload and turns its LEDs on to display the least three significant bits of the received variable value. After displaying the values, the sensor transmits its count variable to the other sensor.

When one of the sensor's communication was halted or switched off, the other sensor displayed the last three LED values it received from the transmitting sensor. These LED values of the receiving sensor match the LED values of the transmitting sensor, further proving back and forth communication between the sensors. The application for this project was built on top of the code for the BlinkToRadio tutorial. Functionality was added to read RSSI values from the CC2420 chip, perform radio to serial operations and light LEDs to show transmission and reception modes. No supporting Java application was used in the TinyOS tutorial package however, this project made use of a tutorial Java application called RssiDemo.

RssiDemo is a tutorial which demonstrates the usage of a Java application to observe RSSI values of received radio signals. In the tutorial, the RssiDemo.java class implements the MessageListener interface. This interface specifies a method called messageReceived which signals the reception of a TinyOS message from serial. The RssiDemo class has a private field called moteIF of type MoteIF. MoteIF provides a java interface that enables the reception and sending of messages via the serial port, a TCP connection or another means of connectivity [52]. It is normally used by creating an instance of it and registering a MessageListener object that is summoned when a message arrives [52]. The constructor of the RssiDemo class creates an instance of a MoteIF and registers a listener. The listener is listening for an RssiMsg. An RssiMsg is a class that extends the Message class and has methods that give provide details on the rssi properties of the message received. The Message class is the base class for encoding and decoding tinyos messages [53]. When the message is received from serial, the messageReceived method is invoked. In the method, the message is cast again to an RssiMsg type. The source of the RssiMsg message is retrieved from the message header via getter functions. The RSSI value is also retrieved from the RssiMsg and printed to the terminal. Functionality was added to read in 100 messages, calculate their rssi values in dBm, by subtracting 45 [50] and log each RSSI value into a file for post processing.

For this project, Alice and Bob send a BlinkToRadio Message between themselves. This message is a struct data type containing a 16 bit variable. The data type of the variable was nx_int16_t. The nx data types ensures interoperability between sensors which are of different endianness [54]. The application used in this project had two components: BlinkToRadioAppC.nc and BlinkToRadioC.nc. BlinkToRadioAppC.nc contained the configuration part of the application and BlinkToRadioC.nc the module portion of the application. Each sensor had components such as BaseStationC in the configuration file that enabled it receive and send messages. A header file called BlinktoRadio.h contained the BlinkToRadio message and enumerated values for timer duration and Active Message type [55] which were used in the configuration and module files. In the module file, a timer was set that fired for the duration specified in the header file. After the timer fired, a BlinkToRadio message was created to be sent over the radio. The send function was called. The OS sent the message and the callback event, sendDone returned after the message was sent. The green LED was programmed to light up if the sendDone function returned.

The other sensor at this time was listening for a message. The RssiMsgIntercept.forward function intercepted the message that was transmitted over the radio by the other sensor. This function retrieves the payload of the received message which is the BlinkToRadioMsg that was transmitted. It casts the payload into a new BlinkToRadioMsg. The RSSI value of the received message is then obtained using the getRssi function and put into the 16 bit RSSI variable that is in the payload of the casted BlinkToRadioMsg. The getRssi function calls another getter method on the CC2420Packet. The CC2420Packet is a nesC interface that provides asynchronous getter and setter methods for packets received or to be sent by the chip [56]. The getRssi function in the interface, gets the RSSI value when a packet is being received. It also provides the RSSI value of an acknowledgement if an acknowledgement was received for a packet that has to be sent. The latter functionality was not utilized in this project [56]. After the RSSI value has been obtained from the chip, the function forwards the contents of the message from radio to serial, that is from the radio buffer to the serial buffer of the sensor. The red LED was programmed to light when the forward function runs signaling that the message was received. After the function ends, the sensor also transmits a message back to the other sensor. The back and forth communication cycle ensues with the LEDs displaying green and red representing transmission and reception.

In order to view the messages and log the data, the RssiDemo Java application must also be run. After the sensors are programmed to run the nesC code, the java program is run in the terminal on each computer at about the same time. On each computer, the program displays the RSSI values of the received messages.

Alice and Bob each transmit a message after their respective timers have fired. If this duration was too long, the communication channel could change significantly and create more errors in the received sequence at each other's end. The transmission time also affected the key rate, which is calculated by dividing the total number of bits transmitted by the duration of transmission. The duration for transmission of a hundred messages between Alice and Bob was measured to determine the actual transmission time. The measurements were done in the RssiDemo application. As the application had a method called messageReceived that was invoked whenever a message was received, it was simple to record the time the first and last messages were received. Time for first message reception was subtracted from time for last message reception. The difference was the total transmission time. The test was performed three times at Alice's and Bob's end and the results averaged.

For a better understanding of the transmission and reception process see Figures 8 and 9.

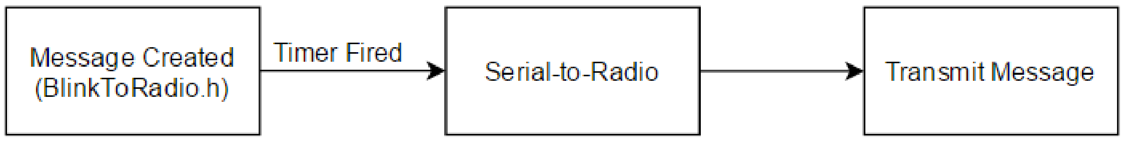


Figure 8 - State diagram of message transmission

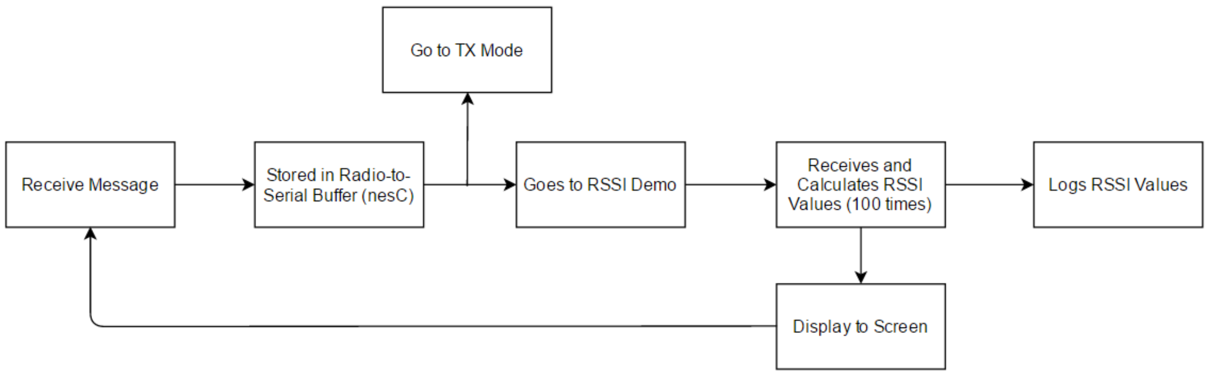

Go to TX Mode
Receive Message
Stored in Radio-to-Serial Buffer (nesC)
Goes to RSSI Demo
Receives and Calculates RSSI Values (100 times)
Logs RSSI Values
Display to Screen

Figure 9 - State diagram of message reception

3.4: RSSI Quantization

Once the sensors were able to communicate back and forth while both continuously transmitting a message and receiving the RSSI measurements from the received signal, the next step was to take these RSSI measurements and quantize them. This was done so that it could be determined how similar the received bits were for both parties. If the bits were similar, with only a few errors, then this would prove that the channel the parties were communicating on was also similar, and the channel estimation could be used to generate a secret key that would only be known to the two communicating parties. Further channel coding and decoding could then be implemented in order to correct these erroneous bits at the receiving end to enhance the channel estimation. If the bits were not similar then this meant that the channel the parties were communicating on was not similar and therefore the channel estimation would not be able to be used for key generation.

With the sensors communicating back and forth the RSSI measurements would then be saved so they could be processed after the sensors were finished communicating. The measurements from both sensors could then be put on one computer and uploaded into a computing software, in this case MATLAB, and the median value for both data sets could be found and used as a threshold. This threshold would then be used to quantize each RSSI value into bits 1 or 0. Any values lying on or below the threshold would be quantized as a 0 and any values lying above the threshold would be quantized as a 1. After both sets of data were quantized, bit sequences for each party would be generated and could be compared.

To compare the bit sequences, the percentage of bits that were different and the percentage of bits that were the same could then be calculated. The closer this percentage was to 100% or 0%, the more the channels were similar and the closer the percentage was to 50%, the more random the channels were. Once these calculations were conducted, a vague idea about the channel randomness could be concluded and further processing to correct bit errors could be completed.

3.5: Error Correction

After Alice and Bob have both transmitted their sequences to the other and quantized their received bits, it is then important to compare the two sequences to see how similar they are. This is done because, in order to properly generate a secret key using the random channel between Alice and Bob, it is crucial that the sequences both Alice and Bob received are the same. If the sequences are not the same, it is important to see how different they are, for example what percentage are the two sequences different. By performing error correction on one of the set of bit sequences, for the most part, will correct the erroneous bits. How much the error correction will actually correct the sequence depends on how different the two sequences are and how robust the coding and decoding schemes are. Once both users have the same bit sequence, it can then be used as the secret key between them. This key will be robust to eavesdropping since it was generated based on the randomness of the channel between Alice and Bob, and therefore, any other channel that is used to try and generate this same key will differ and thus so will the secret key.

3.5.1: (5, 2) Binary Linear Block Channel Coding and Encoding

Channel coding is the introduction of redundancy into the transmitted signal to enable error detection and subsequent correction [57]. The type of coding employed in this project is Linear Block Coding. Linear codes are easy to design and implement. This was a major reason why this coding technique was utilized. A linear code takes the form (n, k) with n and k as integer values and n greater than k. To perform linear coding, sets of k symbols are used to generate sets of n bits. These sets of n bits make up the coded sequence to be transmitted over the air. A generator matrix of size k x n is used to produce the n bits that make up the new coded sequence [57]. In this project a (5, 2) linear code was used on Alice's quantized data. The approach was to apply this coding scheme to generate a new set of 150 bits that would be sent to Bob. He would use this new sequence to detect and correct errors in his quantized values. A 2x5 generator matrix is used in creating the sequence. The generator matrix G is

$$G = \begin{bmatrix} 1\ 0\ 1\ 0\ 1 \\ 0\ 1\ 0\ 1\ 1 \end{bmatrix}. \tag{5}$$

The standard form of this matrix is

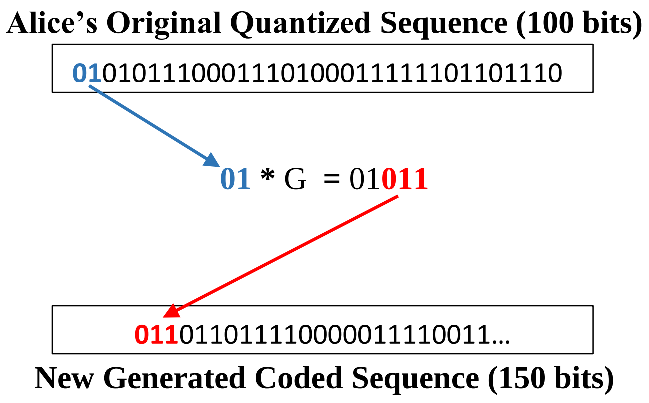
$$G = [I_k|P], \tag{6}$$

where I_k is the k by k identity matrix and P is an arbitrary k by r matrix. For example, with the above generator matrix G, I_k and P are defined as

$$I_k = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \tag{7}$$

$$P = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}. \tag{8}$$

r is the number of redundant bits that are added into the sequence and is found by subtracting k from n. For a better understanding, this coding scheme can be seen below.



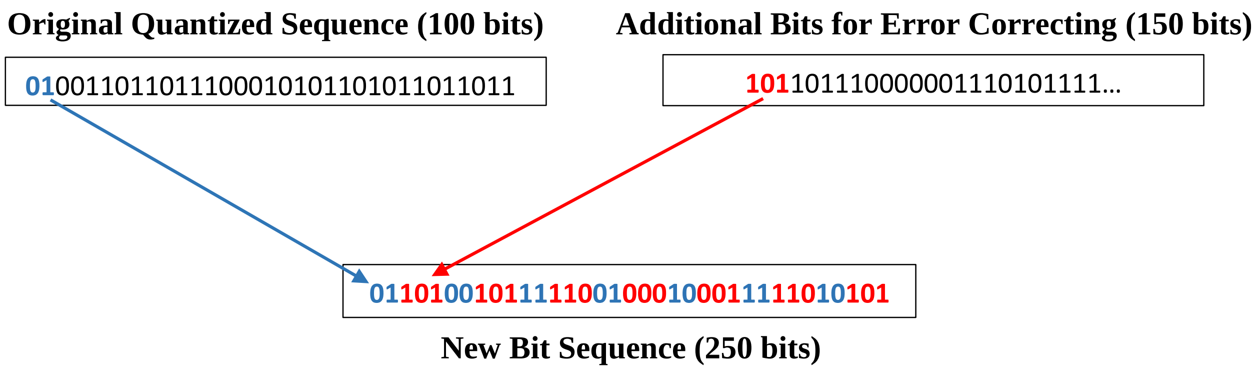
Alice's Original Quantized Sequence (100 bits)
0101011100011101000111111101101110
01 * G = 01011
011011011110000011110011...
New Generated Coded Sequence (150 bits)

To apply this encoding scheme, a simple loop in MATLAB was used to get two bits from Alice's quantized sequence to perform a modulo 2 multiplication with the Generator matrix. The result of the modulo arithmetic was a 5 bit value. The last 3 bits of the result were stored in a new sequence to be transmitted to Bob. The operation was repeated for every two bits until the end of Alice's sequence was reached. At this point, a new sequence of 150 bits was generated. This sequence was to be transmitted to Bob.

The sequence used for error correction could not be sent in the form it was in as no data structure in nesC was able to contain its size. An approach was developed to break the bits into small sequences and then send that sequence over the radio to Bob. In order to break the bits into sequences of equal length, the sequence was zero padded to 160 bits. The 160 bit sequence was divided into ten smaller sequences with 16 bits in each sequence. While testing this approach, it was noted that Bob truncated the 16 bits and took only the last four bits. A solution was to convert each 16 bit to hexadecimal and have four hex values in place of each sequence. Therefore, Alice would transmit 10 sequences each containing 4 hex values. The BlinkToRadioMsg struct for both Alice and Bob was modified to have 10 nx_int16_t variables named bits1a, bit1b, bits 2a, bits2b, all the way to bits5a, and bits5b. The 10 hexadecimal values to be transmitted were coded into the enumerated type declaration in the BlinkToRadio.h header file. They were named BITS1a, BITS1b, BITS2a, BITS2b all the way BITS5a, and BITS5b. In Alice's BlinkToRadioC file, when the timer was fired, the BlinkToRadio message was created and its values were set to the enumerated type values mentioned above. BIT1a's value was put into the bit1a field of the BlinkToRadioMsg continuing through until BIT5b's value was put into the bit5b field. The BlinkToRadioMsg was then transmitted to Bob. Once the bits were defined, Alice began transmitting the 10 error correction bit sequences while Bob was listening for these sequences and only transmitting bit sequences initialized to 0x0000.

3.5.2: (5, 2) Binary Linear Block Channel Decoding

Once the new sequence of bits were coded and Alice transmitted these additional bits to Bob, they were then used to correct Bob's original quantized bit sequence to try and match Alice's bit sequence 100%. The 150 additional bits were received at Bob's end and logged as decimal values in a text file so it could be accessed once the sensors were finished communicating. To correct Bob's original bit sequence, the file that was saved with the decimal values would then be loaded into MATLAB so that the post processing could be completed. First all of the decimal values were converted into their binary equivalents. Since 160 bits had to be sent in order to receive the correct information and only 150 bits were needed, the new bit sequence was indexed so that there were only 150 bits in the sequence. With these 150 bits, the sequence was then indexed so that three bits at a time were taken and appended to the two bits that were indexed from Bob's original sequence to create a new sequence of 250 bits [57]. An example of this can be seen below.


Original Quantized Sequence (100 bits)
Additional Bits for Error Correcting (150 bits)
0100110110111000101011010110110 11
101101111000000111010111 1…
01101001011111001000100011111010101
New Bit Sequence (250 bits)

From this new sequence every 5 bits, a 1x5 array, was taken and multiplied by the parity-check matrix [57], where the parity check matrix is

$$H^T = \begin{bmatrix} 1\ 0\ 1 \\ 0\ 1\ 1 \\ 1\ 0\ 0 \\ 0\ 1\ 0 \\ 0\ 0\ 1 \end{bmatrix}. \tag{9}$$

The standard form of this matrix is

$$H^T = [-P^T | I_{n-k}]^T, \tag{10}$$

where I_{n-k} is the n-k by n-k identity matrix and P^T is the transpose of P. For example, with the above parity-check matrix H^T, I_{n-k} and P^T are

$$I_{n-k} = \begin{bmatrix} 1\ 0\ 0 \\ 0\ 1\ 0 \\ 0\ 0\ 1 \end{bmatrix}, \tag{11}$$

$$P^T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}. \quad (12)$$

This multiplication was conducted using `mod(`$x_1 * H^T$,2), which, in MATLAB, is equivalent to performing binary multiplication, resulting in a 1x3 array. This result is "Syndrome $\mathbf{e}H^T$", and each of these arrays, once computed, were stored in a larger array that contained all values of $\mathbf{e}H^T$ [57]. Every three bits of this array were then taken and compared to the "Syndrome $\mathbf{e}H^T$" column in the Syndrome Table below.

Table 4 - Syndrome table [57]

Syndrome $\mathbf{e}H^T$	Most likely error $\mathbf{e}$
000	00000
001	00001
010	00010
011	01000
100	00100
101	10000
110	11000
111	10010

Once the bits were matched up to the "Syndrome $\mathbf{e}H^T$" column, depending on which sequence occurred, the most likely bit error was able to be determined. Wherever there was a one in the five-bit sequence under the "Most likely error e" column in the syndrome table, is where a bit error had most likely occurred. In this case, only the first two bits were important in order to determine bit errors, therefore, only the first two bits out of the five bits that were indexed, needed to be corrected. As can be seen from the table above, if the syndrome $\mathbf{e}H^T$ was the sequence 000, 001, 010 or 100 then no errors had occurred in the first two bits, which meant no bits had to be corrected. If the bit sequence was 011 then a bit error had occurred in the second bit. To correct the error this same bit sequence of two had to be indexed from Bob's original quantized sequence and the second bit had to be flipped. If the bit sequence was 101 or 111 then a bit error had occurred in the first bit, where to correct this bit, the two-bit sequence from Bob's original sequence had to be indexed and the first bit had to be flipped. Finally, if the sequence was 110 then an error had occurred in both of the first two bits. To correct this, again, the same two-bit sequence from Bob's original sequence had to be indexed and both bits had to be flipped. Once all of the syndrome $\mathbf{e}H^T$ sequences were compared to the syndrome table and the appropriate bits were corrected, Bob now had a new bit sequence with a length of 100 bits. With Bob's new corrected sequence it was again compared against Alice's sequence to ensure that the two were the same.

4. Results and Discussion

This chapter discusses the results that were obtained for the various scenarios tested. These scenarios were when Alice and Bob were both stationary, Alice moving and Bob stationary, Alice, Bob, and Eve stationary, and Alice and Bob stationary with Eve moving. For each of these scenarios the raw RSSI values for each sensor were gathered, quantized, and compared. The quantized data was then corrected using linear coding and the results were plotted and compared.

4.1: Scenario 1: Alice and Bob Remain Stationary

4.1.1: Raw RSSI Measurements

After communications between Alice and Bob was complete, both users had a set of 100 RSSI values that were measured from their received messages. These RSSI values ranged from -41 dBm to -32 dBm. This range did not vary greatly because neither of the users were moving, thus their channels were not changing and instead remained approximately constant. To observe the variations in the data, Alice and Bob's measurements were overlaid in the plot below. Alice's measurements are represented using the solid blue line and Bob's measurements are represented using the purple line.

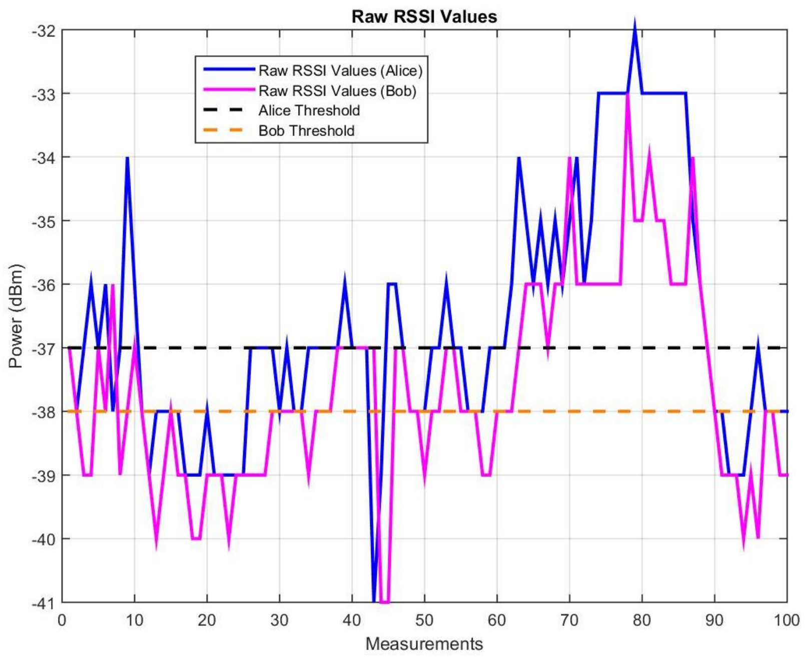


Figure 10 - Received RSSI values at Alice and Bob before quantization

From this graph it can be seen that both sets of data follow the same general curve. This result is promising because it shows that the channel between Alice and Bob share the same characteristics for both directions of communication.

Once the RSSI measurements were plotted, a threshold was set for both Alice and Bob using the median of their sets of data. Each of these thresholds would then be used to quantize the raw RSSI values into a bit sequence consisting of 1's and 0's. Alice's threshold is represented using the black dashed line while Bob's threshold is represented using the orange dashed line. Once each user had their quantized bits sequences, they were then compared.

4.1.2: RSSI Quantization

Each user now had a bit sequence of 100 bits that were then plotted and compared. The graph below shows Alice and Bob's quantized bit sequences overlaid, where Alice's sequence is represented using the solid blue line and Bob's sequence is represented using the purple dashed line.

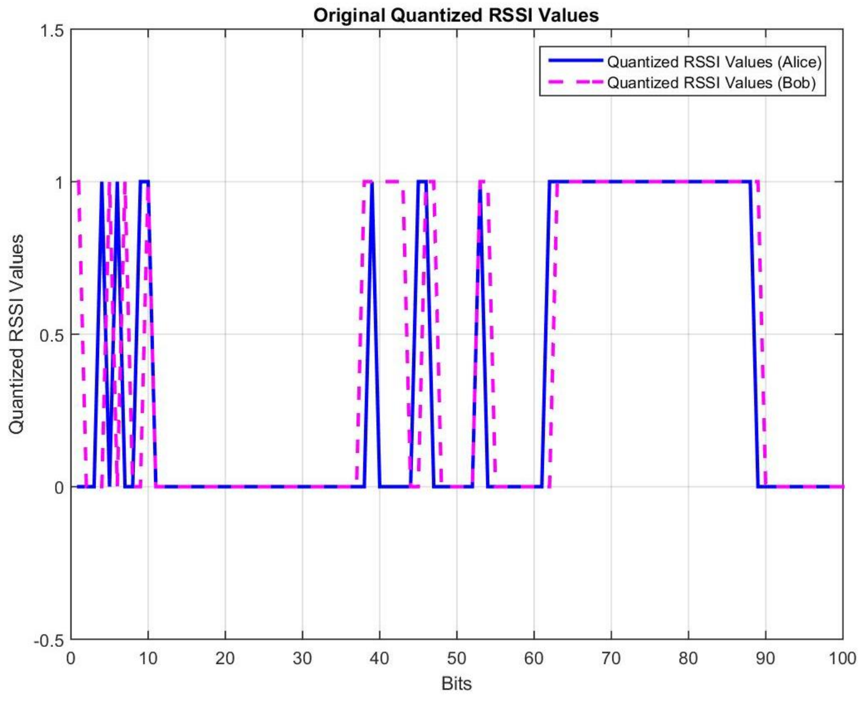


Figure 11 - Quantized RSSI values at Alice and Bob

As can be seen from Figure 11, the quantized bit sequences were fairly similar, but had some differences. Due to these noticeable differences, the sequences were further compared by subtracting the two bit sequences, bit-by-bit respectively, to see for which bits the sequences differed. The absolute value of this result was then plotted and can be seen in Figure 12.

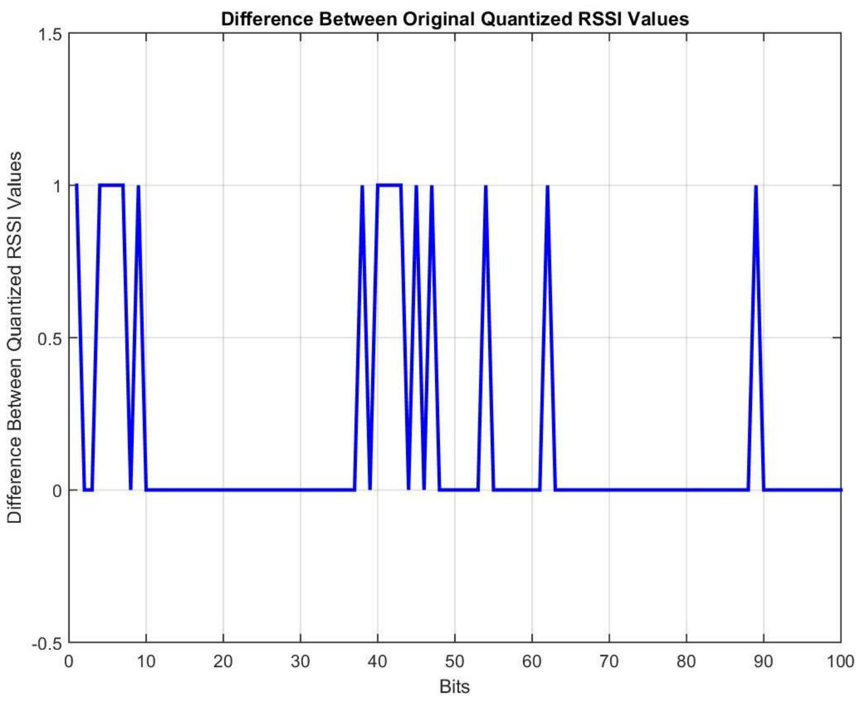


Figure 12 - Difference between Alice and Bob's quantized sequences

From this plot, it can be seen that where peaks occurred at the value 1 was where the sequences were different and where the value was 0 was where the sequences were the same. From this result, the average number of 0's was found to determine the percentage of bits that were the same between the sequences.

It was found that 84 bits were the same between the two sequences and 16 bits were different. This resulted in an 84% similarity between the two sequences. This meant that the two sequences were more deterministic as opposed to random, and thus, the channel between the two users were similar. To further improve the similarity of the sequences, error correcting was then performed and the results were observed.

4.1.3: Bit Sequence after Coding

The MATLAB code (Appendix A) was used on Alice's data to perform a (5,2) Linear Block Coding. 150 bits were generated from the error coding and zero padded to 160 bits. The 160 bits were then converted to hexadecimal. The hexadecimal values transmitted to Bob were 0x0D8C0000, 0x00000050, 0x3002801E, 0xDB6DB6DB, 0x60000000. Each set was broken into two parts and transmitted independently. The format is below:

BITS1a = 0x0D8C
BITS1b = 0x0000
BITS2a = 0x0000
BITS2b = 0x0050
BITS3a = 0x3002
BITS3b = 0x801E
BITS4a = 0xDB6D
BITS4b = 0xB6DB
BITS5a = 0x6000
BITS5b = 0x0000

4.1.4: Bit Sequence after Decoding

After reception of the transmitted error coding sequence, Bob used MATLAB code (Appendix A) to convert the received sequence to binary, remove the zero padding and correct his bit sequence to match that of Alice. The result of Bob's corrected 100 bit sequence was overlaid with Alice's for comparison. The comparison can be seen in Figure 13. Alice's sequence and Bob's corrected sequence fit correctly over each other.

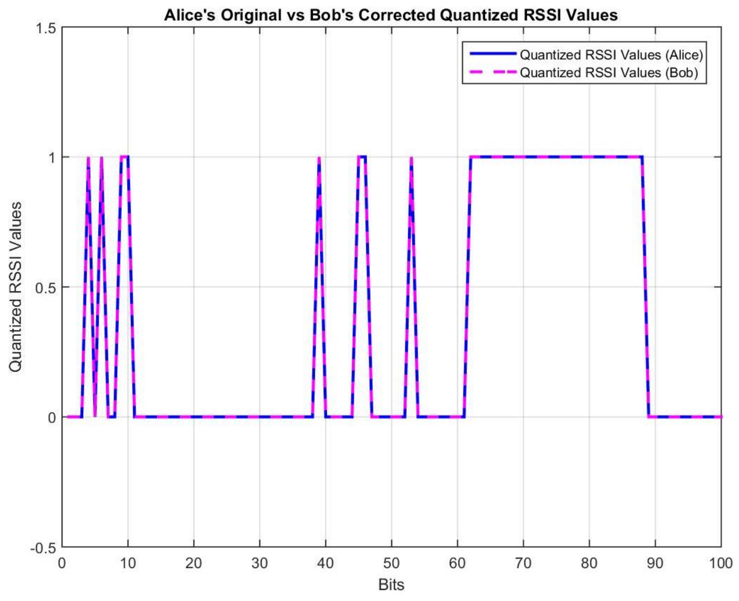


Figure 13 - Quantized RSSI values at Alice and Bob after error correction

The difference of both sequences was then calculated and the result also plotted. Figure 14 represents this difference. It can be observed that there are no peaks for all the bits plotted as compared to Figure 12.

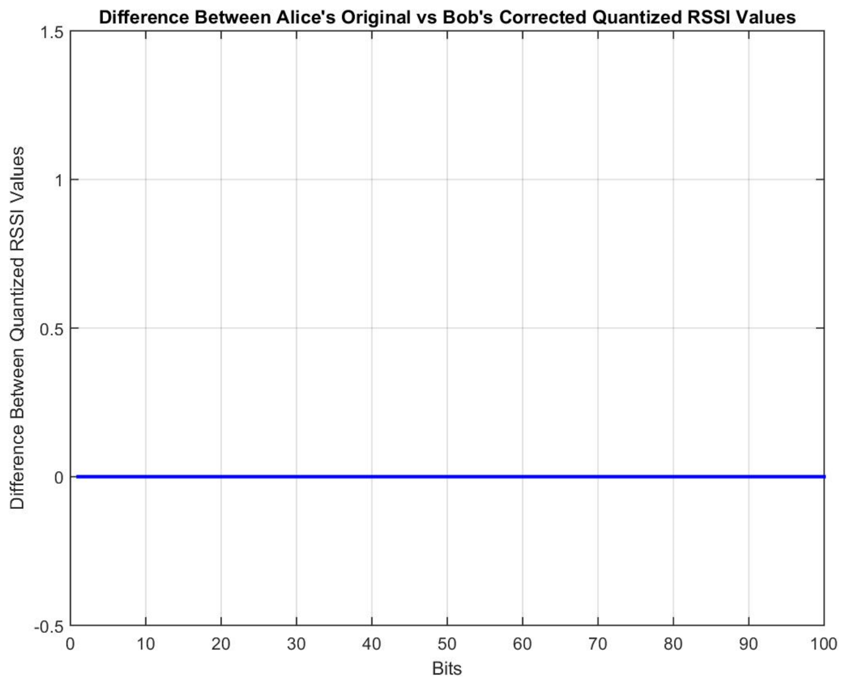


Figure 14 - Difference between Alice and Bob's quantized sequence after error correction

The sum of the difference in Alice's sequence and Bob's corrected sequence was also calculated and seen to be 0. The mathematical and graphical results show that Bob had accurately corrected his bit sequence and removed the 16% disparity between his and Alice's original sequences.

4.2: Scenario 2: Alice moving and Bob Stationary

4.2.1: Raw RSSI Measurements

The raw RSSI measurements were gathered for the next scenario where Alice was moving and Bob was stationary. As can be seen from the graph below, the RSSI measurements were quite different than those gathered from the first test.

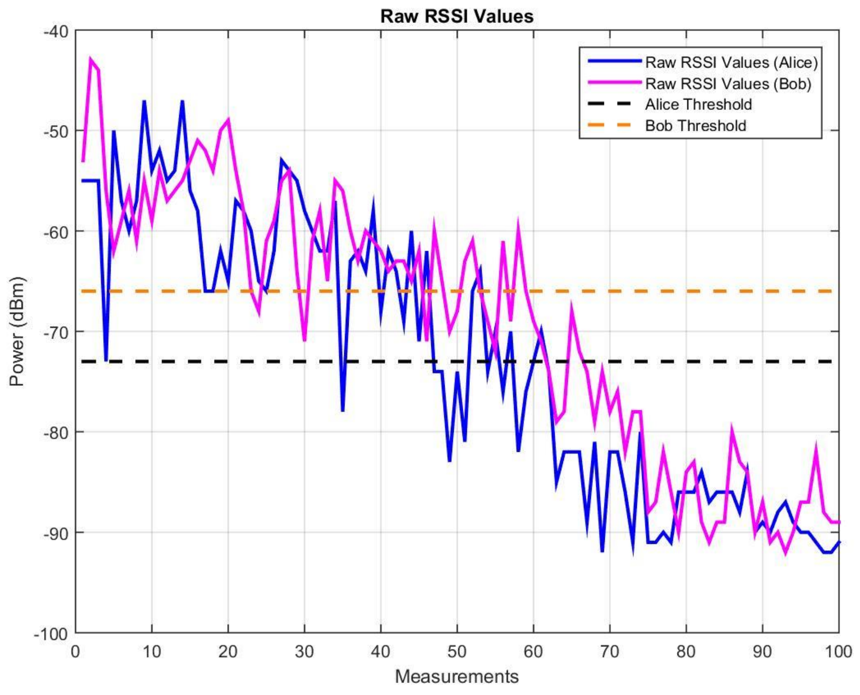


Figure 15 - Received RSSI values at Alice and Bob before quantization

The measurements ranged from about -95 dBm to -40 dBm, which was a much larger range than for the first test. It can also be seen that the RSSI values started off strong and then began to decrease in power. This was because during testing, Alice started off close to Bob and then moved away as communications continued. Both Alice and Bob shared these same characteristics in their RSSI measurements, as this was expected because they were communicating across the same channel.

With the measurements plotted the median values of each of their sets of data was determined in MATLAB. Alice's threshold is represented using the black dashed line and Bob's threshold is represented using the orange dashed line in the graph above. These values were then used as a threshold to quantize Alice and Bob's raw RSSI measurements into bit sequences of 1's and 0's. Once each user had their quantized bit sequences, they were then compared.

4.2.2: RSSI Quantization

Once each set of RSSI measurements were quantized based on the threshold for each user, the bit sequences were then plotted to observe any differences. Both Alice and Bob's bit sequences were overlaid in the plot below, where Alice's bit sequences is represented using the solid blue line and Bob's bit sequence is represented using the purple dashed line.

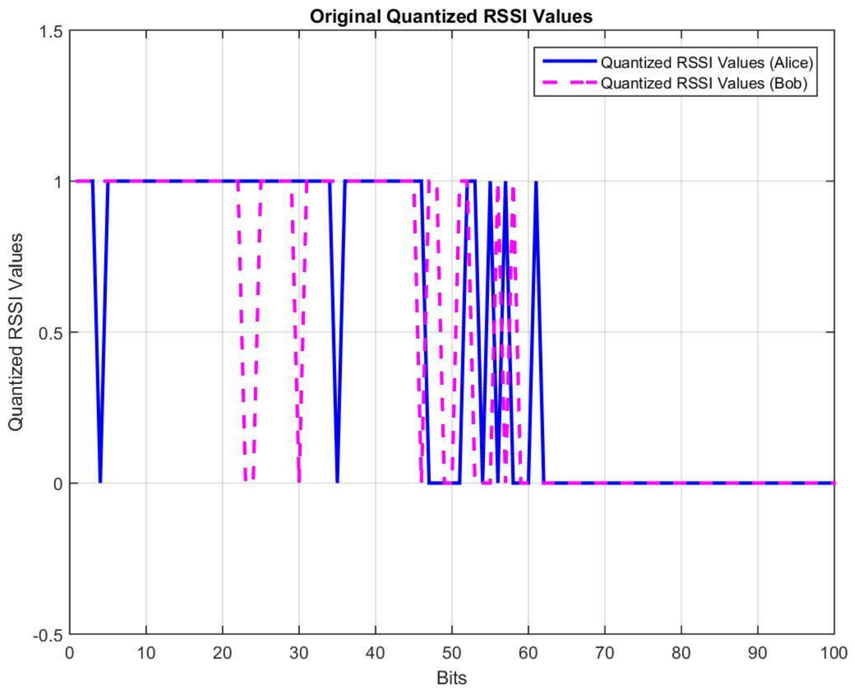


Figure 16 - Quantized RSSI values at Alice and Bob

From this plot it can be seen that the bit sequences did not match up exactly. There were some ambiguities in the quantized sequences that needed to be corrected. To find out how similar the two sequences were, they were subtracted bit-by-bit, respectively, and plotted. This result can be seen in Figure 17.

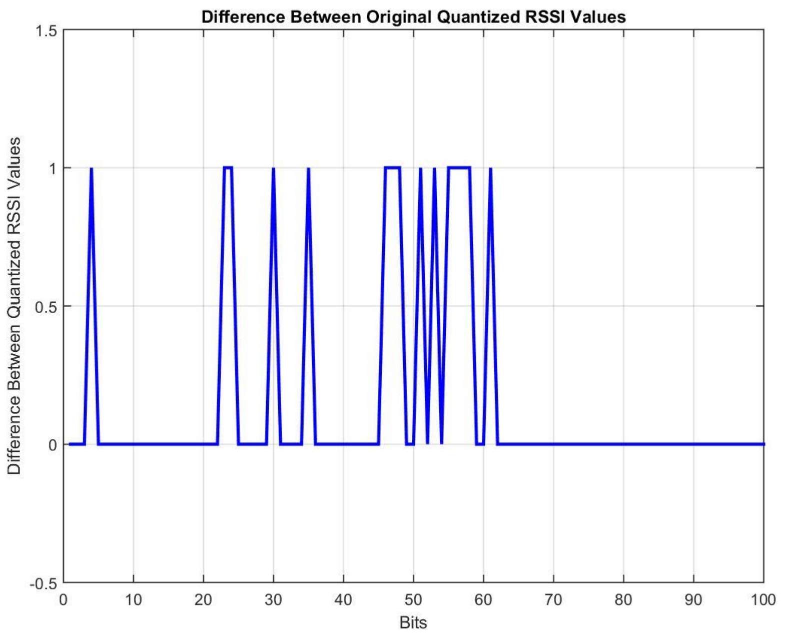


Figure 17 - Difference between Alice and Bob's quantized sequences

From this graph it can be seen that where the peaks formed at a value 1 was where the bit sequences differed, and where the values were 0 was where the bit sequences were the same. Using this result, the percentage of bits that were the same between the two sequences was determined by finding the average number of 0's in this new bit sequence.

In doing this, the bit sequences were found to be 85% similar. This result concluded that, even when the channel is changing between Alice and Bob, they still share a similarity where their bit sequences can be determined from one another. While Alice and Bob's sequences did not match-up 100%, this result proved hopeful that once the sequences were cleaned up they would be the same, thus also concluding that the same channels were used during communications between them. In order to achieve this, error correcting was then performed in order to clean up Bob's sequence. These results were again compared with Alice's original sequence to observe their similarities.

4.2.3: Bit Sequence after Coding

After quantization, the (5,2) linear code was used on Alice's sequence to generate the error correction sequence that would be transmitted to Bob. A 150 bit error correction sequence was generated. This was zero padded to 160 bits and then split into 5, 32 bit sequences. Each of the 5 sequences was converted to their corresponding hexadecimal values. Each of the 5 sequences had 8 hexadecimal values. Each hexadecimal value was split into two before transmission to Bob. The sequence transmitted to Bob was as follows:

BITS1a = 0xD765
BITS1b = 0xB6DB
BITS2a = 0x6DB6
BITS2b = 0xCF6D
BITS3a = 0xB00E
BITS3b = 0xDA28
BITS4a = 0x0000
BITS4b = 0x0000
BITS5a = 0x0000
BITS5b = 0x0000

BITS4a to BITS5b are all zeros. This was because Alice at that period of time was quite far from Bob. This resulted in very low RSSI values. After quantization, these low values were all below the median mark and were therefore zeros. As such, generating an error sequence for that section resulted in zeros.

4.2.4: Bit Sequence after Decoding

After reception of the error correction sequence from Alice, Bob used the MATLAB code (Appendix A) to convert the sequence to binary, remove the zero padding and correct his sequence of bits. The results of his corrected sequence were plotted against Alice's original sequence. Figure 18 represents the comparison. It can be seen in Figure 18 that Alice and Bob's bits overlap exactly, suggesting 100% similarity.

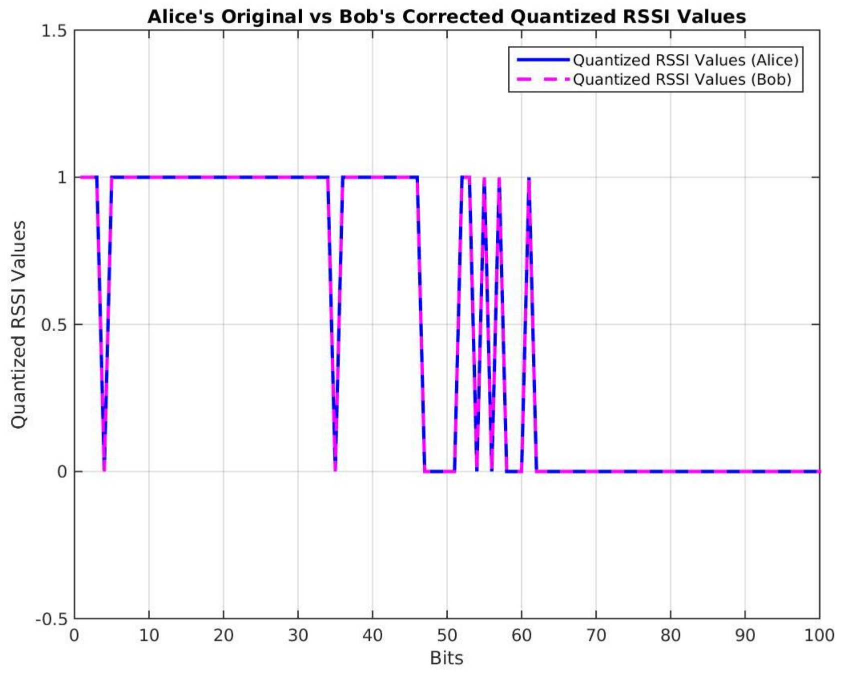


Figure 18 - Quantized RSSI values at Alice and Bob after error correction

Alice’s original and Bob’s corrected sequence were subtracted bit-by-bit from each other and the result was plotted. The plot in Figure 19 shows this result. It can be seen that there were no spikes which would represent a discrepancy at those corresponding bits. The mathematical result of the subtraction came out to 0.

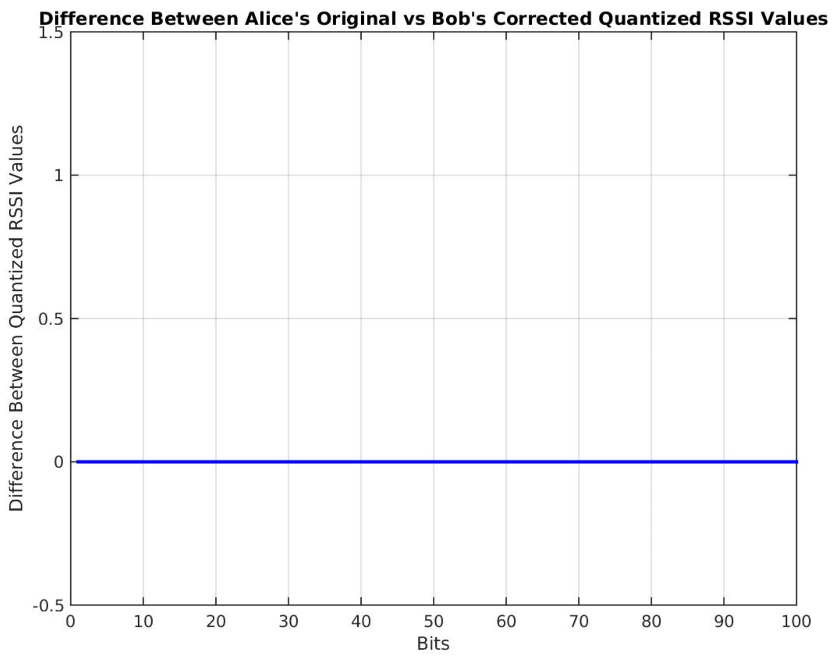


Figure 19 - Difference between Alice and Bob's quantized sequence after error correction

The graphical and mathematical result of the comparison shows that Bob had accurately corrected the 15% discrepancy in his and Alice's original sequences.

4.3: Scenario 3: Alice, Bob, and Eve Remain Stationary

4.3.1: Raw RSSI Measurements

The next test was conducted where Alice and Bob communicated while Eve was eavesdropping on their communications. In this scenario, Alice and Bob both transmitted and received their sequences while Eve remained passive, just listening. All three users remained stationary for these communications. The raw RSSI measurements were gathered at Alice, Bob, and Eve's end and plotted below. Alice's measurements are represented using the solid blue line, Bob's measurements are represented using the solid purple line, and Eve's measurements are represented using the solid green line.

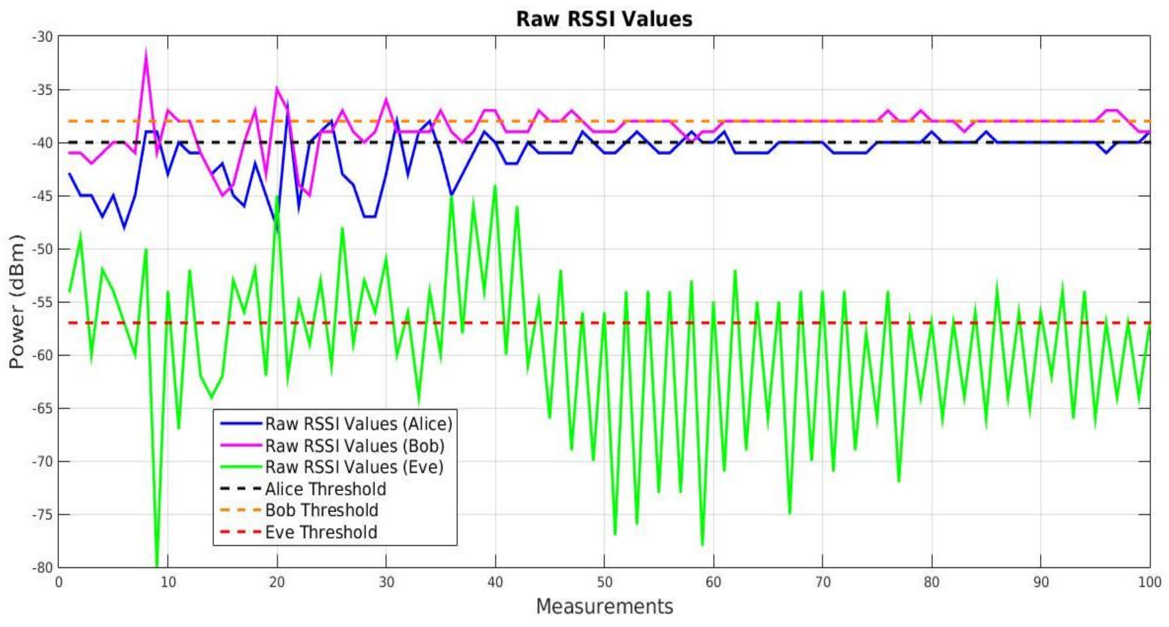


Figure 20 - Received RSSI values at Alice, Bob, and Eve before quantization

As can be seen from Figure 20, for Alice and Bob's measurements, the values ranged from about -32 to -48 dBm while Eve's measurements ranged from about -45 to -80 dBm. From these measurements it can also be seen that Eve's RSSI values were not on the same scale as Alice and Bob's measurements. This is because Eve was placed in a different location as opposed to Alice and Bob, and although Eve still had line-of-sight for each of the users, the channel she was communicating on was different.

Once the data was plotted, the thresholds were determined based on these raw measurements. The median value for each user's set of data was used as the threshold. In the above graph, Alice's threshold is represented using the orange dashed line, Bob's threshold is represented using the black dashed line and Eve's threshold is represented using the red dashed line. Their RSSI values were then quantized to either 1's or 0's based on this threshold.

4.3.2: RSSI Quantization

With each user's set of data quantized, they were then overlaid in Figure 21 to observe similarities and differences in their sequences. Alice's bit sequence was represented using the solid blue line while Bob and Eve's sequences were represented using the purple and green dashed lines, respectively.

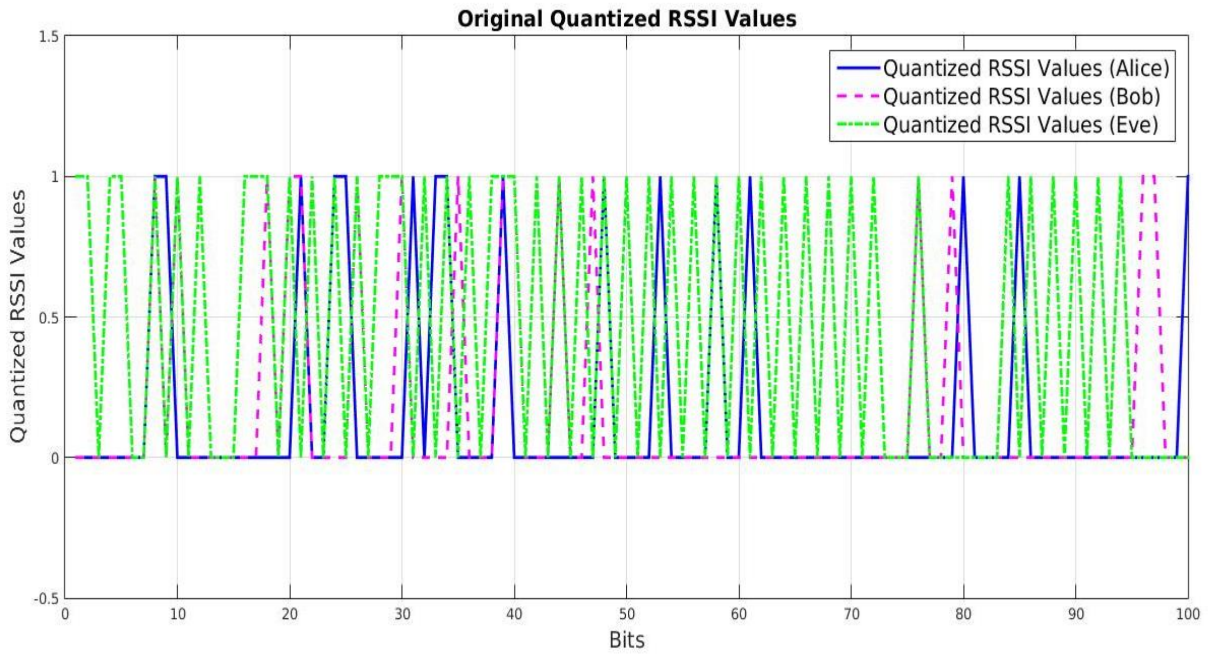


Figure 21 - Quantized RSSI values at Alice, Bob, and Eve

From this graph it can be seen that Eve's bit sequence quantized differently than Alice and Bob's sequences. Eve's sequence also was seen to vary more, changing from 1 to 0 more often than Alice and Bob's sequences. Once these sequences were observed, they were further compared by determining how different each sequence was to one another. To do this each combination of sequences were taken and subtracted from their paired sequence bit by bit, respectively.

First, Alice and Bob's sequences were subtracted from one another to determine how similar they were. Where peaks occurred was where the sequences differed and where zeros occurred was where the sequences were the same. This result can be seen in Figure 22.

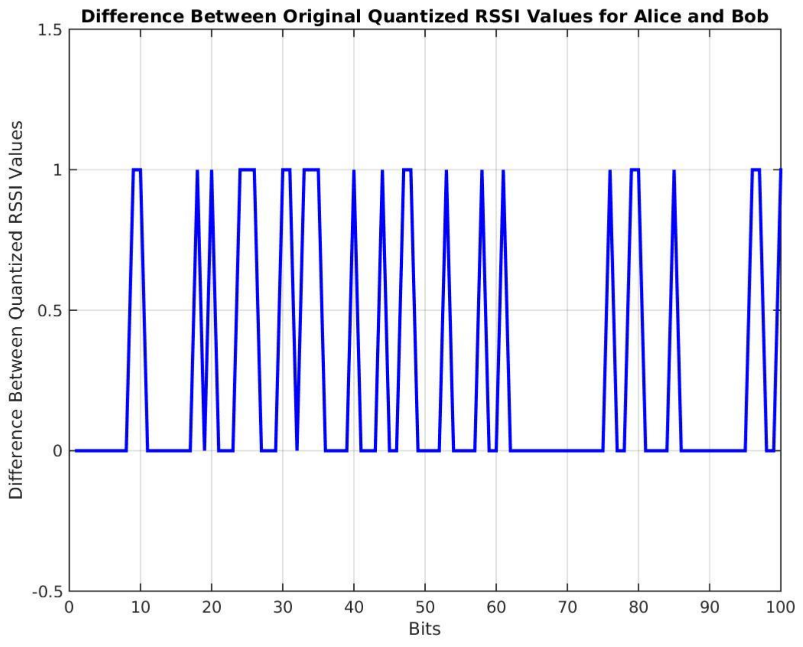


Figure 22 - Difference between Alice and Bob's quantized sequences

Alice and Bob’s sequences were proven to be 74% similar. This showed that the sequences were close to being the same, and after error correcting was performed, the sequences would have a good chance of matching up 100%, thus concluding that the same channel was used in communications.

Next, Alice and Eve’s bit sequences were compared. They were subtracted bit by bit, respectively and the absolute value of the result was plotted. Each bit that resulted in a one was where the sequences differed and each bit that resulted in a zero was where each sequence was the same. This can be seen in Figure 23.

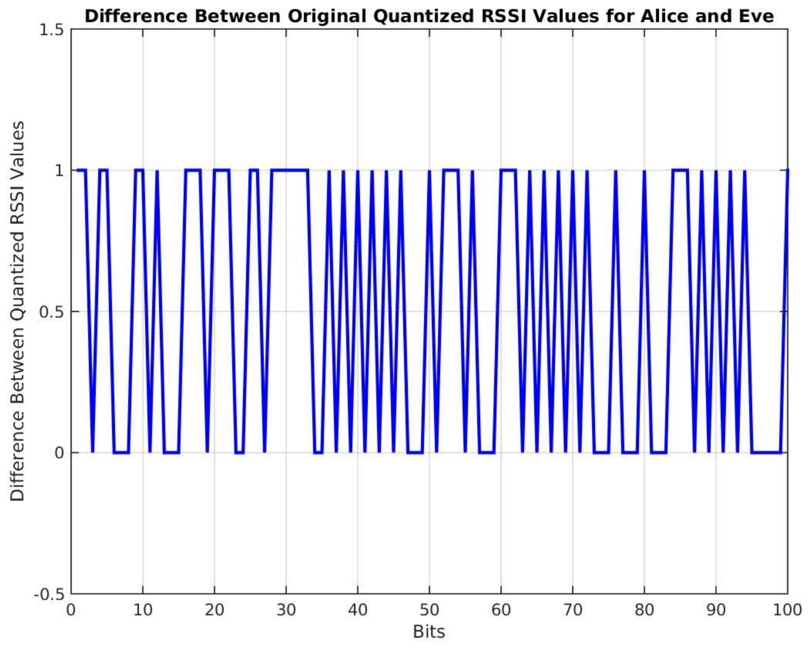


Figure 23 - Difference between Alice and Eve's quantized sequences

From this graph it can be seen that there were many peaks that resulted as opposed to previous comparisons. This is because Alice and Eve’s sequences were only 50% similar, which means that the sequence that Eve observed was random in regards to Alice’s sequence and that the same channel was not used for communications.

Finally Bob and Eve’s sequence was compared and subtracted bit by bit. The resulting sequence was plotted below where, again, peaks represented where the sequences were different and zeros represented where the sequences were the same for that bit. This can be seen in Figure 24.

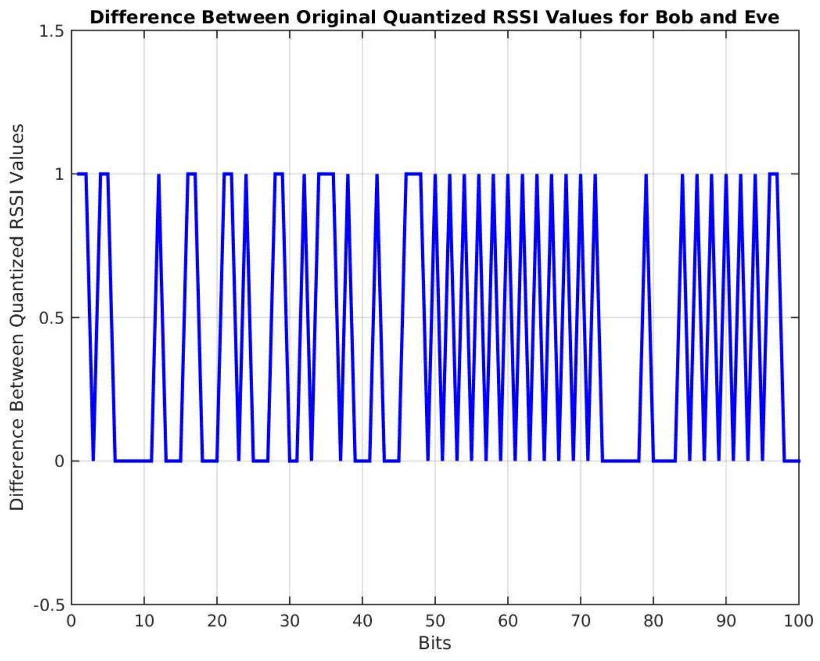


Figure 24 - Difference between Bob and Eve's quantized sequences

From this graph it can be seen that, just as with the comparison between Alice and Eve, the comparison between Bob and Eve’s sequence was also very different, only resulting in a 58% similarity. This too meant that the sequence Eve observed was independent of the sequence that Bob observed on his channel. Since the sequences were random, this proved that different channels were used during communications.

With these results, the next step was to perform error correction on Bob’s sequence in order to clean up the 26% erroneous bits at his end. This would lead to Alice and Bob both having the same sequence as they would then be able to use this sequence as a secret key for secure communications.

4.3.3: Bit Sequence after Coding

In order to correct the 26% error in the original bits of Alice and Bob, the (5,2) linear code was used on Alice's bit sequence. The resulting 150 bit sequence was zero padded to 160 bits. These bit were split into 5, 32 bits and converted to 5, 8 hexadecimal values. Each of the hexadecimal values was split into two before transmission to Bob. 10 sequences were then transmitted to Bob. The error correction sequence transmitted to Bob was as follows

BITS1a = 0x003A
BITS1b = 0x0002
BITS2a = 0xBA05
BITS2b = 0xC050
BITS3a = 0x0302
BITS3b = 0x8628
BITS4a = 0x0000
BITS4b = 0x0302
BITS5a = 0x8000
BITS5b = 0x0C00

4.3.4: Bit Sequence after Decoding

After reception of the corrected sequence from Alice, Bob used MATLAB Code (Appendix A) to convert the sequence to binary, remove the zero padding and correct his sequence of bits. The results of Bob's corrected sequence was plotted against Alice's original sequence and can be seen in Figure 25.

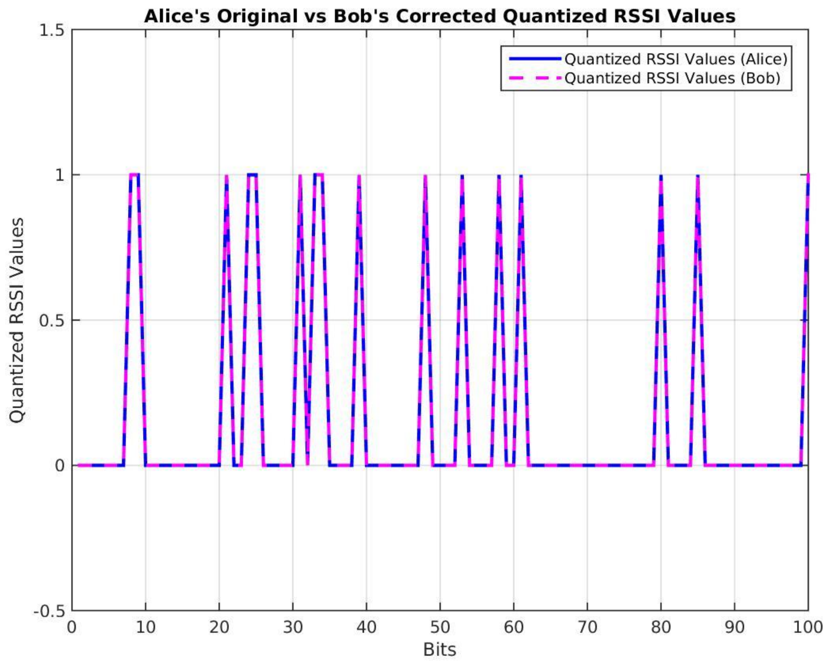


Figure 25 - Quantized RSSI values at Alice and Bob after error correction

The plot shows that Alice's original and Bob's corrected sequence exactly overlay over each other. The difference between both sequences was calculated bit-by-bit and the result plotted. The plot can be seen in Figure 26.

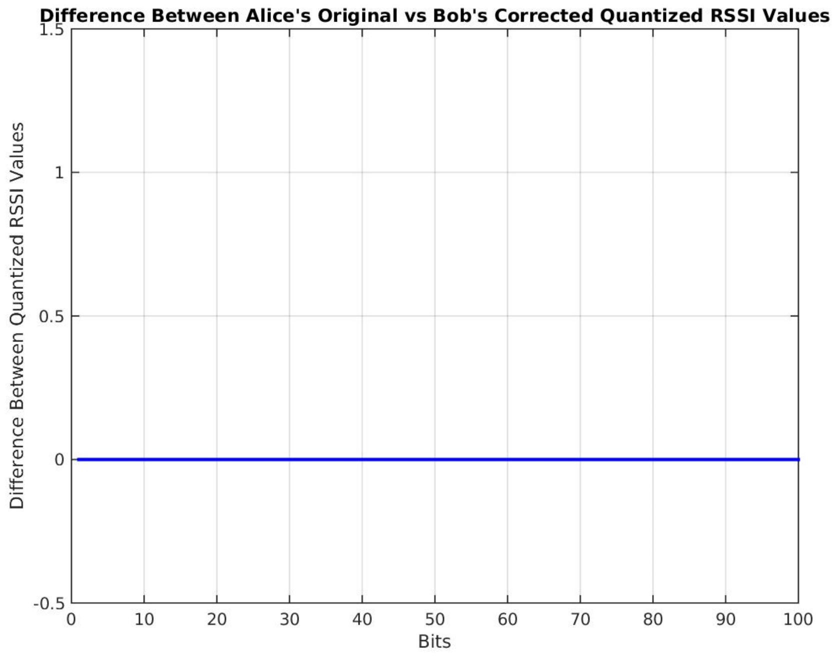


Figure 26 - Difference between Alice and Bob's quantized sequences

The result of the difference between the two sequences came out to be zero. The plot also shows no peaks, suggesting 100% similarity between the two sequences. From the mathematical and graphical result, it can be seen that Bob accurately corrected the 26% error in his original sequence.

4.4: Scenario 4: Alice and Bob Remain Stationary While Eve is Moving

4.4.1: Raw RSSI Measurements

The last test was conducted where Alice and Bob again communicated across a stationary channel while Eve was listening to their communications on a varying channel. While Eve was gathering her set of RSSI measurements she remained passive and was moving away from Alice and Bob. Alice and Bob also gathered their own set of raw RSSI measurements from one another while remaining stationary. All three sets of data can be seen in Figure 27 where Alice’s data is represented using the solid blue line, Bob’s data is represented using the solid purple line, and Eve’s data is represented using the solid green line.

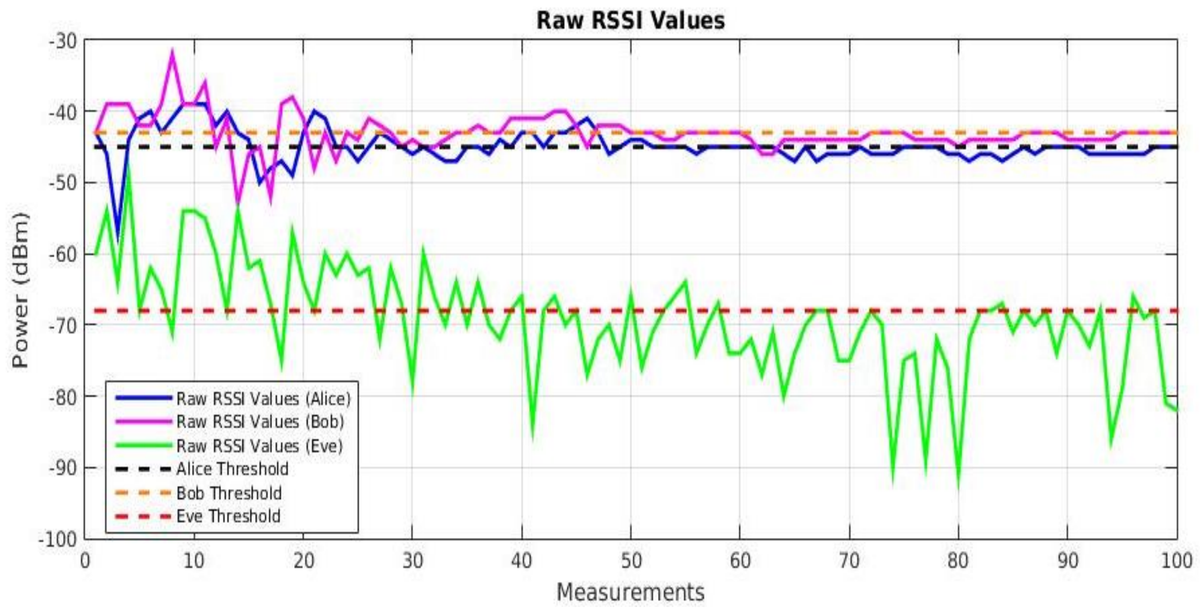


Figure 27 - Received RSSI values at Alice, Bob, and Eve before quantization

From this graph it can be seen that Alice and Bob's measurements ranged from about -30 to -55 dBm while Eve's measurements ranged from about -45 to -95 dBm, a 25 dBm difference compared to a 50 dBm difference. This is because Eve's channel was moving as she was collecting her data, meaning that Eve did not share the same channel that Alice and Bob shared.

With the raw data compared, a threshold was then set for each user based on the median value of their set of data. This threshold was then used to quantize the raw RSSI measurements into 1's or 0's in order to form a unique bit sequence for each user. Alice's threshold is represented using the orange dashed line, Bob's threshold is represented using the black dashed line, and Eve's threshold was represented using the red dashed line in Figure 27. Using these thresholds, each of their bit sequences were formed and further compared.

4.4.2: RSSI Quantization

Once each user had their own bit sequences, they were then overlaid for comparison. This can be seen in Figure 28 where Alice's sequence is represented using the solid blue line, Bob's sequence is represented using the purple dashed line, and Eve's sequence is represented using the green dashed line.

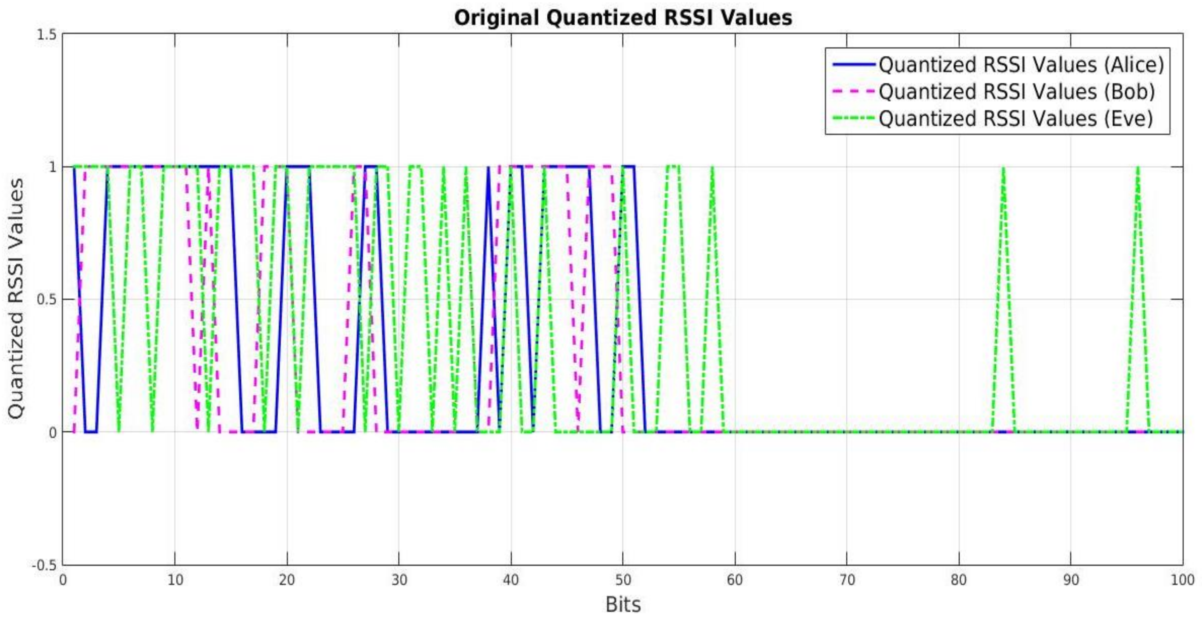


Figure 28 - Quantized RSSI values at Alice, Bob, and Eve

From the above figure it can be seen that all three quantized bit sequences differed, however, there were also some areas where all three sequences were the same. For example, in the beginning, the sequences seemed to differ more so than towards the end of the sequence, where, at around bit 60 to bit 82 the sequences were all the same. This was unexpected because Eve started out being able to see Alice and Bob, similar to Eve's location in the previous test, and then moved away from them as communications continued. Nevertheless, to determine exactly how each bit sequence differed from one another each combination of sequences were compared by subtracting each of their sequences bit by bit, respectively with one another.

Alice and Bob's sequence was first compared. Their bit sequences were subtracted and the absolute value of this result was then plotted. Peaks of one meant that the sequences were different for that particular bit and values of zero resulted when the sequences were the same. This can be seen in Figure 29.

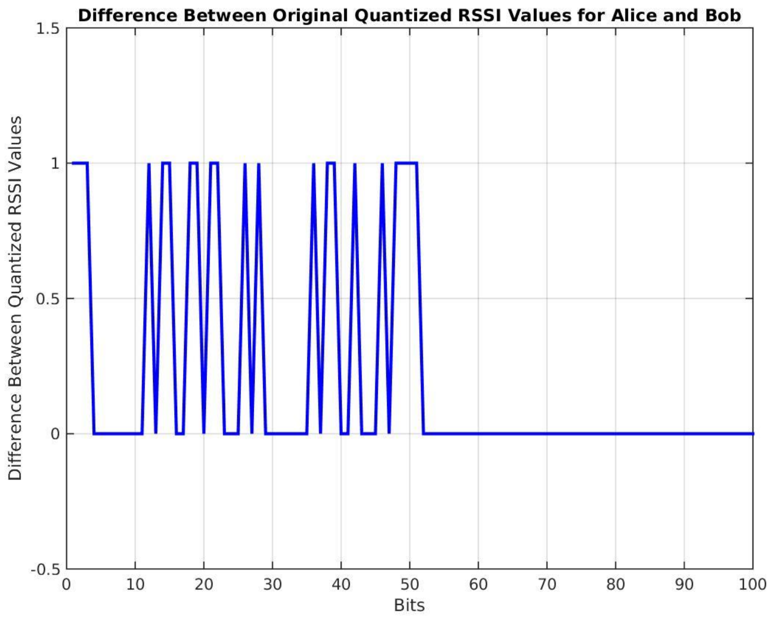


Figure 29 - Difference between Alice and Bob's quantized sequences

From this figure it can be seen that there were not many differences in the two sequences, but when there were variations among the two sequences, it occurred in the first half of the bit sequence. The second half of the bit sequences were the same between them. This resulted in the Alice and Bob’s sequences being 79% similar, which meant that, since their bits were similar their channel was also similar, and after error correction their channel would most likely be 100% similar.

Next, Alice and Eve’s bits were compared in the same fashion. The result was plotted in Figure 30 to observe the differences amongst their bits.

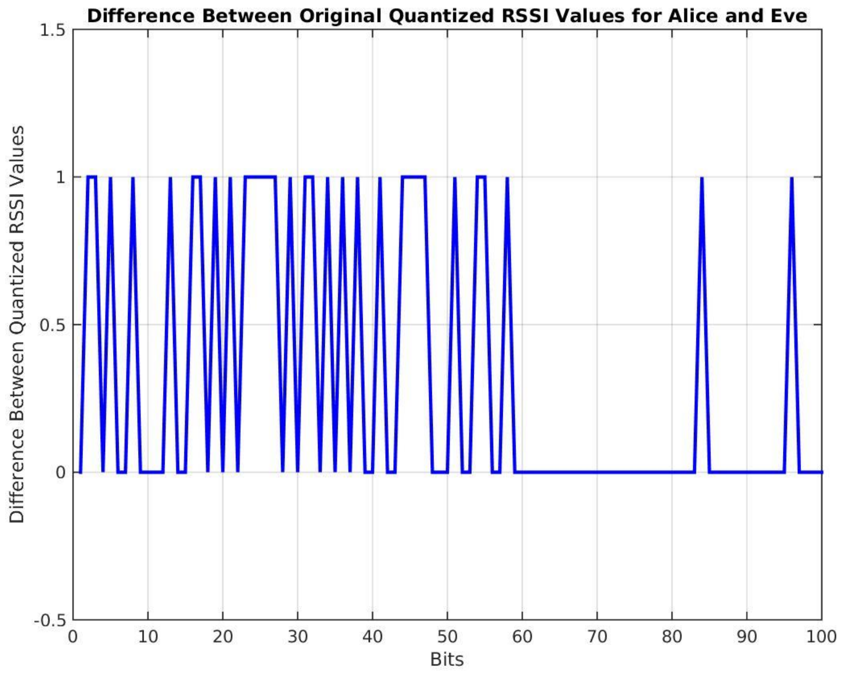


Figure 30 - Difference between Alice and Eve's quantized sequences

From this figure it can be seen that there were many differences between Alice and Eve's sequences, while most of these bits were different in the first half of the sequence. Towards the end of the sequence their bits were very similar, except for two instances were peaks of one occurred at bits 83 and 95. Again, this was unexpected since the distance between Alice and Eve was greater as the communications progressed and the channel was changing more rapidly. These bit sequences were 69% similar, which was a higher percentage as opposed to the previous test. The reason for this higher percentage can clearly be seen in the last half of the bit sequence where these bits were very similar as opposed to previous tests.

Lastly, Bob and Eve's bit sequences were then compared. Once the bits were subtracted respectively, they were plotted to see how the two sequences differed. This plot can be seen in Figure 31.

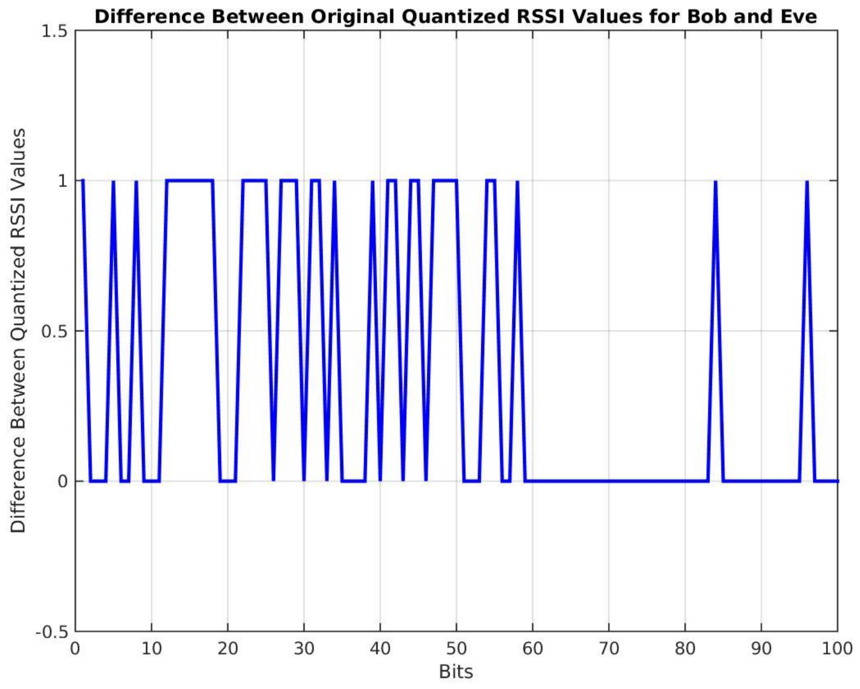


Figure 31 - Difference between Bob and Eve's quantized sequences

As can be seen above, this plot shared a striking resemblance to the comparison between Alice and Eve’s bit sequences. Just as in Figure 30, the bits differed more in the first half of the sequence and were the same in the last half of the sequence. While this was unexpected, it can be confirmed by looking at Figure 28 that the sequences were in fact similar in the last 40 or so bits. With this observation, it was found that Bob and Eve’s sequence was 66% similar.

While Eve’s sequence was more similar to Alice and Bob’s sequence in this set of tests as opposed to in the previous set of tests where the users were stationary, the similarity between them was still more random and could be concluded that Alice and Bob most likely shared the same channel due to their 79% similarity while Eve did not share this same channel. This higher percentage of similarity may have been due to the fact that Eve’s channel was not as constant as in the third testing scenario, thus, resulting in this ambiguity. Error correction was then performed on Bob’s sequence of bits in order to generate the same sequence at both Alice and Bob’s end, cleaning up the 21% of bits that were different.

4.4.3: Bit Sequence after Coding

To correct the 21% disparity between Alice's and Bob's original sequence, the (5,2) linear code was used on Alice's original sequence. The resulting 150 bits was zero padded to 160 bits. The 160 bits were split into 5 sequences, each 32 bits long. Each sequence was converted to hexadecimal. The result was 5 sequences, each having 8 hexadecimal digits. Each hexadecimal sequence was split into two before transmission. The transmitted digits are below

BITS1a = 0xAF6D
BITS1b = 0xB50F
BITS2a = 0x0180
BITS2b = 0x01BB
BITS3a = 0xB574
BITS3b = 0x0000
BITS4a = 0x0000
BITS4b = 0x0000
BITS5a = 0x0000
BITS5b = 0x0000

4.4.4: Bit Sequence after Decoding

After reception of the error correction sequence from Alice, Bob used the MATLAB code (Appendix A) to convert the bits to binary, remove the zero padding and correct his bits. The results of his corrected sequence was plotted against Alice's original sequence. The result can be seen in Figure 32.

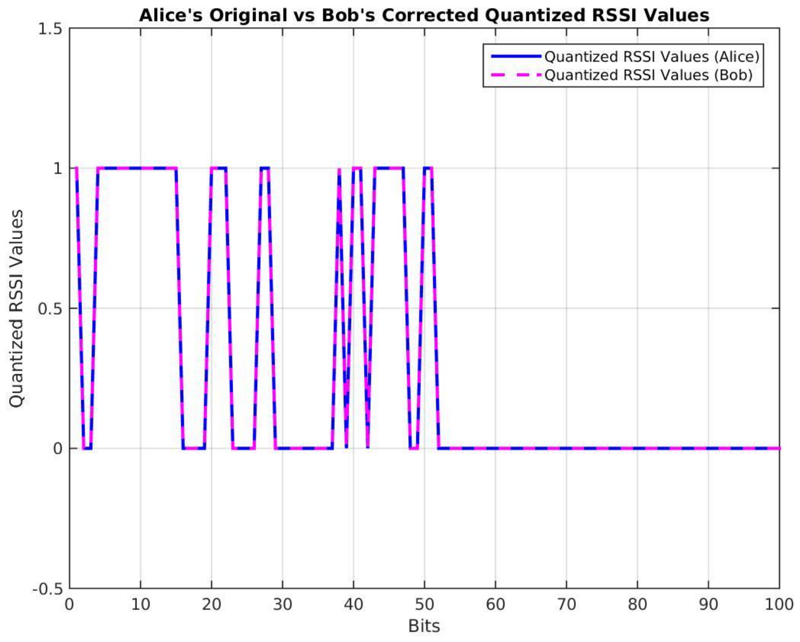


Figure 32 - Quantized RSSI values at Alice and Bob after error correction

It can be observed that both figures overlapped perfectly. Alice's original sequence was subtracted from Bob's corrected sequence bit by bit. This was done to determine if there was any discrepancy between the sequences. The result was found to be 0 and the plot can be seen in Figure 33. The absence of peaks show that both sequences are similar.

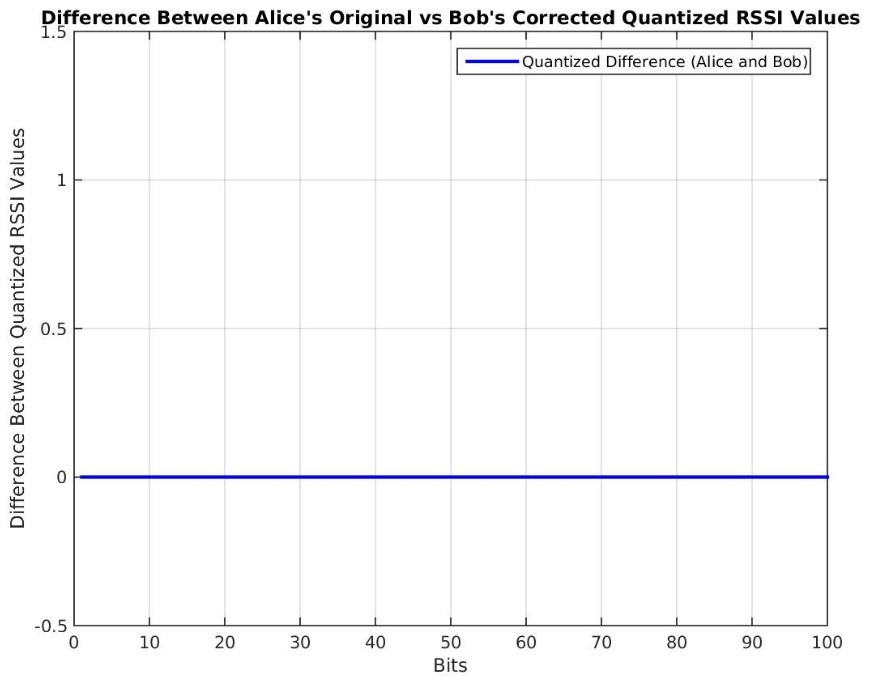


Figure 33 - Difference between Alice and Bob's quantized sequences

The mathematical and graphical results show that Bob was able to accurately correct the 21% disparity in his original sequence compared to Alice's original sequence.

4.5: Testing Error Correction on Eve’s Sequence

The same set of bits that Bob received to correct his sequence was then used to try and correct Eve’s sequence. The same decoding procedure was also used with Eve’s original quantized sequence in order to correct it. Ideally, when this error correction was performed on Eve’s sequence, it should not be able to correct her bit sequence, however, for both scenarios, where Eve was stationary and where Eve was moving, the error correction corrected Eve’s sequence 100% to match Alice’s sequence. This was surprising because for each scenario Eve’s sequence only matched up to Alice’s sequence 50% and 69%, respectively. This coding scheme corrected a sequence that was random, which meant that the coding scheme was too robust. Since Bob’s sequence was still more similar to Alice’s sequence compared to Eve’s sequence, as was expected, this meant that the coding scheme could be less robust and would still be able to correct Bob’s sequence, but not Eve’s. While at first the fact that Eve’s sequence was corrected is alarming, this is actually good because this means that the coding scheme can be less robust and therefore not take up as much of the channel bandwidth as the current coding scheme.

4.6: Transmission Duration and Key Rate

The duration of transmission affects the key rate. For example, if it takes 2 seconds to send 1 bit, then 100 bits would take 200 seconds to send. The key rate would be the number of bits transmitted divided by the total duration for transmission. In this example, the key rate would be 0.5 bits/second. For this project, the actual transmission time at Alice's and Bob's end was calculated and can be seen in Table 5.

Table 5 - Duration of transmission times for each set of 100 messages and the average between the three tests

	Test 1 (s)	**Test 2 (s)**	**Test 3 (s)**	**Average (s)**
Alice	21.114	21.199	21.965	21.426
Bob	24.346	24.289	24.937	24.524

In the BlinkToRadio.h file for Alice and Bob, the timer is set to fire every 250ms. This results in 25s for transmission of 100 messages. The averaged results for Bob match this value but those of Alice show that she transmitted signals much faster. This may be explained by different CPU cycles of the computers representing Alice as compared to that of Bob's.

Using the timer value, the estimated key rate for this project was 100 bits / 25second = 4bits/second. The actual key rate at Alice's end was 100 bits / 21.426second = 4.667 approximately 5 bits/second. The actual key rate at Bob's end was 100 bits / 24.524second = 4.077 approximately 4 bits/second.

5. Conclusion and Future Work

The results obtained for the different scenarios reinforced the fact that communication security based on information theoretic approaches was feasible. After error correction, Alice and Bob had the same sequence, providing a common key to secure their communication. This common key is noticeably different from a key generated at Eve's end. Various modifications and tests were formulated over the course of the project but time constraints hindered their implementation. These tests can be done in future work to reinforce and improve upon the results obtained in the project. They are further discussed below.

In Scenario 4, the high similarity between Eve's sequence and Alice's and Bob's compared to results from Scenario 3, is a phenomenon which should be explored more. When Eve is moving away from Alice and Bob the percentage of similar bit values should be much lower compared to when Eve is stationary. This phenomenon was reflected and justified in Scenario's 1 and 2 where similarity in Alice's and Bob's sequence in Scenario 1 was higher than in Scenario 2. More testing with Eve moving can provide insights into the unusual results obtained for Scenario 4. For Scenario 3 and 4, Eve was introduced into the environment as a passive observer. Future work can include Eve as an active participant, sending out malicious signals to disrupt effective communications between Alice and Bob.

The various scenarios were set up in an indoors environment. Future tests can also be performed in an outdoor environment. This would introduce new factors that were not anticipated in this project. For example, more distortion in channel estimation testing due to moving objects such as vehicles, human beings and animals. As this technology has applications in communications, this change in testing environment would strongly simulate actual usage and real-life application of the technology. Tests can also be conducted in different weather conditions. A scenario where Alice and Bob are members of a network of sensors can also be tested. The effects of other sensors in the network on the generation of a unique secret key between Alice and Bob can also be explored. Different scenarios can be tested to observe how multiple communications in the network affect the channel estimation between Alice and Bob. The strength of the (5,2) linear code can also be tested on the different scenarios that have been outlined.

During testing, the error correction sequence was applied to Eve's quantized bits in Scenario 3 and 4. It was observed that Eve successfully corrected her bits and produced a sequence that was 100% similar to Alice's original sequence and Bob's corrected sequence. These results show that the (5,2) linear code used for error correction was very robust and contained more information than was necessary. Future tests can have a less robust linear code that produces a sequence that successfully corrects Bob's bits but is unsuccessful for Eve's bits in the event that she obtains the error correction sequence. Such a code is one where less n bits are generated for every k bits. For example, a (3,2) linear code, where $n = 3$ bits are produced for every $k = 2$ bits as compared to $n = 5$ bits produced for every $k = 2$ bits in a (5,2) linear code.

The duration of transmission can also be reduced to increase the key rate. Different timer values can be used to manipulate the transmission time to obtain larger key rates.

TelosB sensors were used to implement functionality for Alice, Bob and Eve. Future work can use USRPs in place of the TelosB sensors. USRPs would provide more control over the specifications for the entire communication process as compared to TelosB sensors which already provide such functionality.

References

[1] *Dictionary.com*. Dictionary.com, n.d. Web. 15 Jan. 2016. <http://dictionary.reference.com/browse/wireless-technology?jss=1>.

[2] Web. <http://www.ctia.org/about-us>.

[3] "CTIA Wireless Industry Survey." N.p., n.d. Web. <http://www.ctia.org/docs/default-source/Facts-Stats/ctia_survey_ye_2014_graphics.pdf?sfvrsn=2>.

[4] Smith, Eric. "Global Broadband and WLAN (Wi-Fi) Networked Households Forecast 2009-2018." 30 Oct. 2014. Web. 29 Nov. 2015. <https://www.strategyanalytics.com/access-services/devices/connected-home/consumer-electronics/reports/report-detail/global-broadband-and-wlan-%28wi-fi%29-networked-households-forecast-2009-2018#.VlpnD-Qy1oA>

[5] "Census, Population Data." Washington Information Directory 2014–2015 (2014): 297. Web.

[6] *Security Features in Wireless Environment.* N.p.: n.p., n.d. PDF. <http://www.artechhouse.com/uploads/public/documents/chapters/imai_520_CH03.pdf.>

[7] Akyildiz, Ian F., and Ismail H. Kasimoglu. "Wireless Sensor and Actor Networks: Research Challenges." *Wireless Sensor and Actor Networks: Research Challenges*. Elseveir B.V., 2004. Web. 16 Nov. 2015. <http://www.sciencedirect.com/science/article/pii/S1570870504000319.>

[8] "Internet of Things Global Standards Initiative." ITU. Web. 30 Nov. 2015. <http://www.itu.int/en/ITU-T/gsi/iot/Pages/default.aspx>.

[9] The President's National Security Telecommunications Advisory Committee. "NSTAC Report to the President on the Internet of Things." Web. 30 Nov. 2015. <http://www.dhs.gov/sites/default/files/publications/IoT%20Final%20Draft%20Report%2011-2014.pdf>.

[10] "Wi-Fi." Wi-Fi. Web. 29 Nov. 2015. <https://www.abiresearch.com/market-research/product/1021330-wi-fi/>.

[11] Beal, Vangie. "Wi-Fi." What Is (IEEE 802.11x)? A Webopedia Definition. Web. 29 Nov. 2015. <http://www.webopedia.com/TERM/W/Wi_Fi.html>.

[12] Mitchell, Bradley. "Why WEP Keys Used to Be Cool but Aren't Very Useful Anymore." About.com Tech. Web. 30 Nov. 2015. <http://compnetworking.about.com/od/wirelessfaqs/f/wep_keys.htm>.

[13] Tews, Erik, Ralf-Philipp Weinmann, and Andrei Pyshkin. "Breaking 104 Bit WEP in Less Than 60 Seconds." Information Security Applications Lecture Notes in Computer Science (2007): 188-202. Web.

[14] Tews, Erik, and Martin Beck. "Practical Attacks against WEP and WPA." Proceedings of the Second ACM Conference on Wireless Network Security - WiSec '09 (2009). Web.

[15] "Sophos Reveals London's WiFi Security Issues with 'Warbiking' Experiment." Sophos Reveals WiFi Security Issues. Web. 29 Nov. 2015. <https://www.sophos.com/en-us/press-office/press-releases/2012/09/sophos-reveals-wifi-security-issues.aspx>.

[16] Web. <https://www.nsa.gov/ia/_files/security_configuration/iad_wireless_security_recommendations.pdf>.

[17] Web. <https://wigle.net/stats#mainstats>.

[18] Messer. "Securing a Wired and Wireless Network – CompTIA A+ 220-802: 2.5 « Professor Messer IT Certification Training Courses." *Professor Messer IT Certification Training Courses*. Messer Studios, 23 May 2013. Web. 23 Dec. 2015. <http://www.professormesser.com/free-a-plus-training/220-802/securing-a-wired-and-wireless-network/>.

[19] Greenemeir, Larry. "TJX Now Largest Data Hack Ever." *InformationWeek*. UBM Tech Brands, 2006. Web. 28 Nov. 2015. <http://www.informationweek.com/tj-maxx-data-theft-likely-due-to-wireless-wardriving/d/d-id/1054964>.

[20] Greenemeir, Larry. "TJX Now Largest Data Hack Ever." *InformationWeek*. UBM Tech Brands, 2006. Web. 28 Nov. 2015. <http://www.informationweek.com/tjx-now-largest-data-hack-ever/d/d-id/1053590>.

[21] Espiner, Tom. "Wi-Fi Hack Caused TK Maxx Security Breach | ZDNet." *ZDNet*. CBS Interactive, 2007. Web. 28 Nov. 2015. <http://www.zdnet.com/article/wi-fi-hack-caused-tk-maxx-security-breach/>.

[22] Schwartz, Mathew J. "Wardriving Burglars Hacked Business Wi-Fi Networks." *InformationWeek: Dark Reading*. UBM Tech Brands, 2011. Web. 28 Nov. 2015. <http://www.darkreading.com/attacks-and-breaches/wardriving-burglars-hacked-business-wi-fi-networks/d/d-id/1100324>.

[23] Roy, Arnab, Anupam Datta, Ante Derek, and John C. Mitchell. "Inductive Trace Properties for Computational Security." (n.d.): n. pag. PDF. <http://seclab.stanford.edu/pcl/papers/rddm-IndTraceProp.pdf>.

[24] *Fresh Horses A Computational Approach to Security*. N.p.: Computer Science Department at Wellesley College. PPT. <http://cs.wellesley.edu/~cs310/lectures/compute_security_slides_handouts.pdf>.

[25] Yin, Zhuo, and Junyuan Zeng. *Computational Security: Public Key Encryption*. N.p.: n.p., 8 Feb. 2009. PDF. <http://isis.poly.edu/courses/cs6903-s09/Lectures/lecture3.pdf>.

[26] "Cryptography." *Wikipedia*. Wikimedia Foundation, n.d. Web. 10 Nov. 2015. <https://en.wikipedia.org/wiki/Cryptography>.

[27] Pass, Rafael, and Abhi Shelat. *A Course in Cryptography*. N.p.: Pass/shelat, 2010. PDF. <http://www.cs.cornell.edu/courses/cs4830/2010fa/lecnotes.pdf>.

[28] Hsu, Jeremy. "Theory Lowers the Speed Limit for Quantum Computing." IEEE Spectrum. IEEE Spectrum, 2015. Web. 15 Nov. 2015. <http://spectrum.ieee.org/tech-talk/computing/hardware/new-calculation-lowers-speed-limit-for-quantum-computing>.

[29] Shikata, Junji. "Formalization of information-theoretic security for key agreement, revisited." Information Theory Proceedings (ISIT), 2013 IEEE International Symposium on. IEEE, 2013.

[30] "One-time Pad." *One-time Pad*. Web. 30 Nov. 2015. <http://www.fact-index.com/o/on/one_time_pad.html>.

[31] Liu, Yanpei, Stark C. Draper, and Akbar M. Sayeed. "Exploiting Channel Diversity in Secret Key Generation from Multipath Fading Randomness." N.p., 1 July 2012. PDF. 25 Nov. 2015. <http://arxiv.org/pdf/1107.3534.pdf>.

Figure 1 "CTIA Wireless Industry Survey." N.p., n.d. Web. <http://www.ctia.org/docs/default-source/Facts-Stats/ctia_survey_ye_2014_graphics.pdf?sfvrsn=2>.

[32] "Received Signal Strength Indication." *Wikipedia*. Wikimedia Foundation, 7 Nov. 2015. Web. 23 Jan. 2016. <https://en.wikipedia.org/wiki/Received_signal_strength_indication>.

[33] "Understanding RSSI." *Trying to Understand RSSI Levels and How to Read Them*. MetaGeek, LLC, 2015. Web. 23 Jan. 2016. <http://www.metageek.com/training/resources/understanding-rssi.html>.

[34] "What Are Broadcasting Power, RSSI and Other Characteristics of Beacon's Signal?" *Estimote Community Portal*. Estimote, Inc., 2014. Web. 23 Jan. 2016. <https://community.estimote.com/hc/en-us/articles/201636913-What-are-Broadcasting-Power-RSSI-and-other-characteristics-of-beacon-s-signal->.

[35] Zeng, Kai, Daniel Wu, An Jack Chan, and Prasant Mohapatra. "Exploiting multiple-antenna diversity for shared secret key generation in wireless networks." In *INFOCOM, 2010 Proceedings IEEE*, pp. 1-9. IEEE, 2010.

[36] Liu, Hongbo, Jie Yang, Yan Wang, Yingying Chen, and Can Emre Koksal. "Group Secret Key Generation via Received Signal Strength: Protocols, Achievable Rates, and Implementation." *Mobile Computing, IEEE Transactions on* 13, no. 12 (2014): 2820-2835.

[37] Shi, Lu. "Secure wearable systems with wireless phy-layer information." PhD diss., UNIVERSITY OF ARKANSAS AT LITTLE ROCK, 2015.

[38] Web. <http://www.advanticsys.com/faq/>.

[39] Khurri, Andrey, Dmitriy Kuptsov, and Andrei Gurtov. "On application of host identity protocol in wireless sensor networks." *Mobile Adhoc and Sensor Systems (MASS), 2010 IEEE 7th International Conference on*. IEEE, 2010.

[40] Web. <http://tinyos.stanford.edu/tinyos-wiki/index.php/FAQ>.

[41] Web. <http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html>.

[42] Gay, David, Philip Levis, Robert Von Behren, Matt Welsh, Eric Brewer, and David Culler. "The NesC Language: A Holistic Approach to Networked Embedded Systems." *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation - PLDI '03* (2003): n. pag. *Http://www.tinyos.net/papers/nesc.pdf*. Web.

[43] "Software Defined Radio." *Cognitive Radio Networks* (n.d.): 41-58. *Wireless Innovation Forum.* Web. 23 Jan. 2016. http://www.wirelessinnovation.org/assets/documents/SoftwareDefinedRadio.pdf

[43] Web. < www.tinyos.net>.

[44] Web. < http://gnuradio.org/redmine/projects/gnuradio/wiki>.

[45] "Introduction." TinyOS Tutorial Lesson 1: TinyOS Component Model. N.p., n.d. Web. 11 Feb. 2016. <http://www.tinyos.net/dist-2.0.0/tinyos-2.x/doc/html/tutorial/lesson1.html>.

[46] "Modules and State." TinyOS Tutorial Lesson 2: Modules and the TinyOS Execution Model. N.p., n.d. Web. 11 Feb. 2016. <http://www.tinyos.net/dist-2.0.0/tinyos-2.x/doc/html/tutorial/lesson2.html>.

[47] "The Ubuntu Story." *About Ubuntu*. Canonical Ltd., 2016. Web. 11 Feb. 2016. <http://www.ubuntu.com/about/about-ubuntu>.

[48] "TinyOS Installation Guide." *Advanticsys Tech Resources*. N.p., 20 Dec. 2013. Web. 11 Feb. 2016. <http://www.advanticsys.com/wiki/index.php?title=TinyOS_Installation_Guide>.

[49] "TinyOS Tutorial #1 - How to Install TinyOS on Ubuntu." *YouTube*. YouTube, 23 Dec. 2012. Web. 11 Feb. 2016. <https://www.youtube.com/watch?v=AJYjy4bSaHw>.

[50] Web. <http://www-mtl.mit.edu/Courses/6.111/labkit/datasheets/CC2420.pdf>.

[51] IEEE std. 802.15.4 - 2003: Wireless Medium Access Control (MAC) and Physical Layer (PHY) specifications for Low Rate Wireless Personal Area Networks (LR-WPANs) http://standards.ieee.org/getieee802/download/802.15.4-2003.pdf

[52] "MoteIF." MoteIF. N.p., n.d. Web. 11 Feb. 2016. <http://www.tinyos.net/dist-2.0.0/tinyos-2.x/doc/javadoc/net/tinyos/message/MoteIF.html>.

[53] "Message." Message. N.p., n.d. Web. 11 Feb. 2016. <http://www.tinyos.net/dist-2.0.0/tinyos-2.x/doc/javadoc/net/tinyos/message/Message.html>.

[54] "CS263r - Harvard University - Assignment #1." CS263r - Harvard University - Assignment #1. N.p., n.d. Web. 10 Feb. 2016. <http://www.eecs.harvard.edu/~mdw/course/cs263/assn1.html>.

[55] "Introduction." Lesson 3: Mote-mote Radio Communication. N.p., n.d. Web. 11 Feb. 2016. <http://www.tinyos.net/dist-2.0.0/tinyos-2.x/doc/html/tutorial/lesson3.html>.

[56] Web. <http://www.tinyos.net/tinyos2.x/doc/nesdoc/micaz/ihtml/tos.chips.cc2420.interfaces.CC2420Packet.html>.

[57] Johnson, C. Richard., William A. Sethares, and Andrew G. Klein. "Coding/Channel Coding." Software Receiver Design: Build Your Own Digital Communications System in Five Easy Steps. Cambridge: Cambridge UP, 2011. 328-332. Print.

[58] Ling, San, and Chaoping Xing. *Coding Theory: A First Course*. Cambridge, UK: Cambridge UP, 2004. Print.

[59] Roman, Steven. *Coding and Information Theory*. New York: Springer-Verlag, 1992. Print.

Appendix A

All MATLAB code can be found in the MATLAB folder.

Alice:
This code can be found in the Alice folder inside the MATLAB folder.

quantizeBits_Alice.m (Same code is used for Bob and Eve where the names are replaced.)

```
% Quantize Alice's bits to get sequence of 100 bits
%% Load RSSI Values Collected from Sending and Receiving Messages
RssiVals_Alice = load('RawRssiVals_Alice_Scenario3.txt');
%% Perform Quantization of Bits
n = 1:100; % number of collected RSSI values
% find median value of RSSI values, to be used as threshold for quantization
medAlice = median(RssiVals_Alice);
% create empty array to store quantized bits (1 or 0)
quantizedRssiVal_Alice = zeros(length(n),1);
% perform quantization, if RSSI value lies below or on threshold then it
% will be quantized as 0 and if it lies above threshold it will be
% quantized as a 1
for numSameQuantVal=1:length(n)
       if RssiVals_Alice(numSameQuantVal,1) <= medAlice
              quantizedRssiVal_Alice(numSameQuantVal,1) = 0;
       else
              quantizedRssiVal_Alice(numSameQuantVal,1) = 1;
       end
end
```

genBits.m

```
% Generate sequence of bits for (5,2) block coding
%% Run quantizeBits_Alice
% run quantizeBits_Alice to get Alice's bit sequence after sending 100
% messages back and forth. Alice's sequence will be used to generate the
% additional 150 bits for coding to send back to Bob in order to correct
% his original quantized bits.
quantizeBits_Alice;
quantizedRssiVal_Alice = quantizedRssiVal_Alice'; % uses Alice's quantized bits for
coding
%% Generate error correction sequence with Alice's original sequence
G = [1 0 1 0 1; 0 1 0 1 1]; % generator matrix
% create empty matrices to store coded bits
coded150bits = [];
for i = 1:2:length(quantizedRssiVal_Alice);
       % index through quantized bits
       % grabs 2 bits every time and uses these bits to generate new coded
       % sequence of 5 bits
       tempBitSeq = quantizedRssiVal_Alice(i:i+1);
       % uses the 2 bits and performs binary arithmetic to generate new 5 bits
       tempNewSeq = mod(tempBitSeq*G,2);
       % creates matrix of newly generated sequence of 150 bits to send to Bob
       coded150bits = [coded150bits tempNewSeq(3:5)];
end
```

```
% zero pad sequence for transmission to Bob
% these are the 160 bits that Alice will send to Bob to correct his sequence
coded160bits = [coded150bits zeros(1,10)]
```

Bob:
This code can be found in the Bob folder inside the MATLAB folder.

decodeBits_Bob.m (The same code was used to try and decode Eve's sequence as well where the names were replaced.)

```
% Decode bits at Bob's end
%% Decoding of transnmitted error-correcting sequence of bits
quantizeBits_Bob; % get Bob's quantized values
toBinary; % converts transmitted hex values to binary and stores them as a string
% at Bob's end, he can just load file that he receives from sensor
bitSequence160 = binArray160; % array from toBinary script
bitSequence150 = bitSequence160(1:150); % removing the zero padding
BobsSeq250bits = [];
quantizedRssiVal_Bob = num2str(quantizedRssiVal_Bob)'; % changes quantized array to
string to match the data type of result from toBinary
quantizedRssiVal_Bob(isspace(quantizedRssiVal_Bob)) = []; % removes empty spaces
%% BITS ARE STRING SO CHANGE FROM HERE DOWN
j=1;
for i = 1:2:length(quantizedRssiVal_Bob);
       % Bob will take every 3 bits from the trasnmitted sequence and append to
       % every 2 bits from his original quantized sequence.
       BobsSeq250bits = [BobsSeq250bits quantizedRssiVal_Bob(i:i+1)
       bitSequence150(j:j+2)];
       j = j + 3;
end
%% Run Decoding on Received Bits at Bob's End
Ht = [1 0 1; 0 1 1; 1 0 0; 0 1 0; 0 0 1]; % parity-check matrix
e_Ht_array = [];
for i = 1:5:length(BobsSeq250bits)
       tempBits = BobsSeq250bits(i:i+4); % index every 5 bits from Bob's sequence
       e_Ht_temp = mod(tempBits*Ht,2); % Syndrome eHt (3 bits)
       e_Ht_array = [e_Ht_array e_Ht_temp]; % all of the eHt's
end
e_Ht_array;
% Syndrome_Table
% 000 00000
% 001 00001
% 010 00010
% 011 01000
% 100 00100
% 101 10000
% 110 11000
% 111 10010
BobInd = 0; % used to index through Bob's original quantized sequence
for j = 1:3:length(e_Ht_array)
       e_Ht = e_Ht_array(j:j+2); % index each e_Ht (3 bits)
       % comparing e_Ht to Syndrome eHt and then flipping bit where there is
       % an error. Only comparing the first 2 bits of "most likely error e", in
       % Syndrome table. Where bit is 1 is where error has most likely
       % occurred.
       if (e_Ht == [0 0 0])
```

```
            % no error in first 2 bits
        elseif(e_Ht == [0 0 1])
            % no error in first 2 bits
        elseif(e_Ht == [0 1 0])
            % no error in first 2 bits
        elseif(e_Ht == [0 1 1])
            % error in 2nd bit
            quantizedRssiVal_Bob(j-BobInd+1) =
            num2str(mod(char(quantizedRssiVal_Bob(j-BobInd+1)+1),2));
        elseif(e_Ht == [1 0 0])
            % no error in first 2 bits
        elseif(e_Ht == [1 0 1])
            % error in 1st bit
            quantizedRssiVal_Bob(j-BobInd) = num2str(mod(char(quantizedRssiVal_Bob(j-
            BobInd)+1),2));
        elseif(e_Ht == [1 1 0])
            % error in both bits
            quantizedRssiVal_Bob(j-BobInd+1) =
            num2str(mod(char(quantizedRssiVal_Bob(j-BobInd+1)+1),2));
            quantizedRssiVal_Bob(j-BobInd) = num2str(mod(char(quantizedRssiVal_Bob(j-
            BobInd)+1),2));
        elseif(e_Ht == [1 1 1])
            % error in 1st bit
            quantizedRssiVal_Bob(j-BobInd) = num2str(mod(char(quantizedRssiVal_Bob(j-
            BobInd)+1),2));
        end
        BobInd = BobInd + 1; % update index
end
% print out these values to command window
quantizedRssiVal_Bob
```

toBinary.m

```
% Open file and read contents. Then convert to binary
% open text file with hex values stored in it
fileId = fopen('AliceCoded3Bits_Scenario3.txt');
binArray160 = []; % array will hold binary conversion of hex values
for i = 1:5
        file = fgetl(fileId);            % get lines of file
        % if end of file then exit while loop and close file
        if file == -1
            fclose(fileId);
            break
        elseif strcmp(file,' ')          % empty line
            fclose(fileId);
            break
        end
        bin = hex2bin2(upper(file));     % convert hex value to binary
        binArray160 = [binArray160 bin]; % store in array of binary values
end
fclose(fileId);
```

hex2bin2.m (This was code was open source and found at this website: http://matlaboratory.blogspot.co.uk/2015/05/converting-hexadecimal-to-binary.html)

```
% converts a value from hex to binary
% code obtained from http://matlaboratory.blogspot.co.uk/2015/05/converting-
hexadecimal-to-binary.html
function bin = hex2bin2(hex)
        % Make sure input is an uppercase string
        hex = upper(num2str(hex));
        if exist('strrep', 'builtin')
                hex = strrep(hex, '0x', '');
        else
                % Strrep not available
                if strcmp(hex(1:2), '0x');
                        hex(1:2)='';
                end
        end
        % Separate each digit
        hex = cellstr(hex');
        % Define lookup table
        % (:,1)=dec, (:,2)=hex, (:,3)=bin
        dhbT= {...
        '0', '0', '0000'; ...
        '1', '1', '0001'; ...
        '2', '2', '0010'; ...
        '3', '3', '0011'; ...
        '4', '4', '0100'; ...
        '5', '5', '0101'; ...
        '6', '6', '0110'; ...
        '7', '7', '0111'; ...
        '8', '8', '1000'; ...
        '9', '9', '1001'; ...
        '10', 'A', '1010'; ...
        '11', 'B', '1011'; ...
        '12', 'C', '1100'; ...
        '13', 'D', '1101'; ...
        '14', 'E', '1110'; ...
        '15', 'F', '1111'; ...
        };
        % For each digit
        for h = 1:length(hex)
                % Find matching row in hex column (2)
                row = strcmp(hex{h}, dhbT(:,2));
                % Replace hex value with bin value
                hex{h} = dhbT{row, 3};
        end
        % Recombine digits
        bin = cell2mat(hex');
```

Other files in this folder include *quantizeBits_Bob.m*.

Eve:
Eve's code can be found in the Eve folder inside the MATLAB folder. These files includes *quantizeBits_Eve.m* and *decodeBits_Eve.m*.

Results and Plots:
This code can be found in the Results and Plots folder inside the MATLAB folder.

errorCorrectionCheck.m

```
%% to check that Alice and Bob have the same bits after error correction
quantizeBits_Alice; % get Alice's quantized bits
decodeBits_Bob;     % get's Bob's corrected bits
% decodeBits_Eve;   % get's Eve's corrected bits
% converting Bob's or Eve's corrected bits from string to int
quantizedIntRssiVal_Bob = zeros(1,length(quantizedRssiVal_Bob));
% quantizedIntRssiVal_Eve = zeros(1,length(quantizedRssiVal_Eve));
for i = 1:length(quantizedRssiVal_Bob)
       quantizedIntRssiVal_Bob(i) = str2num(quantizedRssiVal_Bob(i));
       % quantizedIntRssiVal_Eve(i) = str2num(quantizedRssiVal_Eve(i));
end
quantizedRssiVal_Alice = quantizedRssiVal_Alice'; % changing to row vector
difference_AB = sum(quantizedIntRssiVal_Bob - quantizedRssiVal_Alice)   % show
differences in Alice and Bob
% difference_AE = sum(quantizedIntRssiVal_Eve - quantizedRssiVal_Alice) % show
differences in Alice and Eve
```

plotsOriginalRssiValues.m: Plots raw RSSI data as well as original quantized sequences for Alice, Bob, and Eve.

plotsCorrectedRssiValues.m: Plots the quantized bit sequences after Bob's sequence has been corrected for Alice and Bob. There are sections commented out that if uncommented will plot Eve's data as well.

Appendix B

Steps for TinyOS Installation on Ubuntu:

- Open terminal
- Go to the website below and **complete Step 1** under “Ubuntu Linux Environment”: http://www.advanticsys.com/wiki/index.php?title=TinyOS_Installation_Guide
- Run "sudo -s" and login in as "root" user simply by typing your password after running “sudo -s” command in terminal
- Continue with the rest of part 2 and 3 in website
- In part 4, navigate to “tinyos-main” folder, or whatever you named your main folder with the TinyOS code and type the command “sudo gedit tinyos.sh” and this will open a blank template for you.
  - If you cannot create a new file in a certain directory, it most likely means some of the folders are locked. In this case navigate to the directory right outside that folder and type the command “chmod 777 <folder name>” where <folder name> is replaced with the name of the folder you want to unlock. Same works for unlocking files.
  - Inside the empty template, copy and paste the following code below, where <local-tinyos-path> is replaced with the complete path to your main TinyOS folder.

```
# Here we setup the environment
# variables needed by the tinyos
# make system

export TOSROOT="<local-tinyos-path>"
export TOSDIR="$TOSROOT/tos"
export
CLASSPATH=$CLASSPATH:$TOSROOT/support/sdk/java:$TOSROOT/support/sdk/java/tinyo
s.jar:.
export MAKERULES="$TOSROOT/support/make/Makerules"
export PYTHONPATH=$PYTHONPATH:$TOSROOT/support/sdk/python

echo "setting up TinyOS on source path $TOSROOT"
```

  - For example replace <local-tinyos-path> with /home/user/Desktop/tinyos-main
    - Make sure this is the full path
- At this stage, switch to the YouTube video and continue from time 9:15. The "tinyos.sh" file you have just created is used at this point.
  - Link to YouTube video: https://www.youtube.com/watch?v=AJYjy4bSaHw

- At time 11:52, if you have issues with running the command "sudo tos-install-jni", it may be that you have not installed some tools. Run, "apt-get install tinyos-tools" and then try running "sudo tos-install-jni" again.
    - If this still does not work make sure you have java installed and then try running the command “sudo apt-get update”.
- Finish through with the video
- If you ever get out of terminal and go back in, ALWAYS RUN “sudo -s” to make you the root user.

First Transmission of 100 Bits:

Alice:

1. Go to this directory: tinyos-main-Alice → apps → tutorials → BlinkToRadio → BlinkToRadio2
2. Follow the steps below:
    - In this directory run *make telosb install, 2*
    - This code initially transmits the 100 messages to Bob to come up with a secret key.
3. Next go to this directory: tinyos-main-Alice → apps → tutorials → BlinkToRadio → java
4. Follow the steps below:
    - In this directory run *java RssiDemo -comm serial@/dev/ttyUSBX:telosb* where the X is replaced with the USB port your sensor is plugged into. If you run *motelist* in the terminal you can see this.
    - This code receives the 100 messages that Bob is sending to Alice, prints them to the terminal, and stores them in a text file to access later.

Bob:

5. Go to this directory: tinyos-main-Bob → apps → tutorials → BlinkToRadio → BlinkToRadio2
6. Follow the steps below:
    - In this directory run *make telosb install, 1*
    - This code initially transmits the 100 messages to Alice to come up with a secret key.
7. Next go to this directory: tinyos-main-Bob → apps → tutorials → BlinkToRadio → java
8. Follow the steps below:
    - In this directory run *java RssiDemo -comm serial@/dev/ttyUSBX:telosb* where the X is replaced with the USB port your sensor is plugged into. If you run *motelist* in the terminal you can see this.
    - This code receives the 100 messages that Alice is sending to Bob, prints them to the terminal, and stores them in a text file to access later.

Eve:

9. Go to this directory: tinyos-main-Eve → apps → tutorials → RssiDemo → RssiBase
10. Follow the steps below:
    - In this directory run *make telosb install, 3*
    - This code only receives messages, doesn't transmit anything.

11. Next go to this directory: tinyos-main-Eve → apps → tutorials → RssiDemo → java
12. Follow the steps below:
    - In this directory run *java RssiDemo -comm serial@/dev/ttyUSBX:telosb* where the X is replaced with the port your sensor is plugged into. If you run *motelist* in the terminal you can see this.
    - This code receives 100 messages that are transmitted between Alice and Bob, prints them to the terminal, and stores them in a text file to access later.

Error Correcting:

Alice:

1. Go to this directory: tinyos-main-Alice → apps → tutorials → ErrorCoding → ErrorCoding2
2. Follow the steps below:
    - In this directory run *make telosb install, 2*
    - This code transmits the 10 hex values (160 bits) for error correction to Bob.
    - Don't need to run the java file for this because Alice is only transmitting this sequence, not receiving anything.

Bob:

3. Go to this directory: tinyos-main-Bob → apps → tutorials → ErrorCoding → ErrorCoding2
4. Follow the steps below:
    - In this directory run *make telosb install, 1*
    - This code sends all 0s to Alice and receives the sequence that she is sending.
5. Then go to this directory: tinyos-main-Bob → apps → tutorials → ErrorCoding → java
6. Follow the steps below:
    - In this directory run run *java RssiDemo -comm serial@/dev/ttyUSBX:telosb* where the X is replaced with the USB port your sensor is plugged into. If you run *motelist* in the terminal you can see this.
    - This code receives the 10 hex values (160 bits) for error correction that Alice is sending, prints them to the terminal, and stores them in a text file to access later.