UNIVERSITY OF NAIROBI

FACULTY OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND INFORMATION ENGINEERING

FINAL YEAR PROJECT REPORT
PROJECT INDEX: PRJ 075

SECURE RF CODED COMMUNICATION

STUDENT NAME: **MBOGHO BILL MGHENYI**
REGISTRATION NUMBER: **F17/81902/2017**
SUPERVISOR: **PROF. V.K. ODUOL**
EXAMINER: **Dr. P.O. AKUON**

This project report is submitted in partial fulfilment of the requirements for the award of the Degree of Bachelor of Science in Electrical and Electronics Engineering from the University of Nairobi

Submitted on 26^{th} of May 2022

DECLARATION OF ORIGINALITY

UNIVERSITY OF NAIROBI
FACULTY OF ENGINEERING
DEPARTMENT OF ELECTRICAL AND INFORMATION ENGINEERING
COURSE NAME: Bachelor of Science in Electrical and Electronics Engineering
NAME OF STUDENT: MBOGHO BILL MGHENYI

REGISTRATION NUMBER: F17/81902/2017
PROJECT: SECURE RF CODED COMMUNICATION

1. I understand what plagiarism is and I am aware of the University policy in this regard
2. I declare that this final year project report is my original work and has not been submitted elsewhere for examination, award of a degree or publication. Where other people's work or my own work has been used, this has properly been acknowledged and referenced in accordance with the University of Nairobi requirements.
3. I have not sought or used the services of any professional agencies to produce this work.
4. I have not allowed, and shall not allow anyone to copy my work with the intention of passing it off as his/her own work.
5. I understand that any false claim in respect of this work shall result in disciplinary action, in accordance with University anti-plagiarism policy.

Signature:..... **Date**:.....
Approved by:

Supervisor:PROF. ODUOL

Signature:..... **Date**:.....
Examiner: Dr. AKUON

Signature:..... **Date**:.....

DEDICATION

To God, my family and friends for being with me through this educational journey.

ACKNOWLEDGEMENT

I would like to acknowledge my family for providing a supportive environment all through my University education and for also providing financial assistance where they could and helping me to test the components. I would also like to thank my friends for offering their support, especially when I had bugs that they knew how to fix, special mention to Alvyne Mwaniki for helping me with the implementation of some of the libraries I used.

I would also like to appreciate my supervisor, Prof. Oduol, for guidance in the project implementation. I would like to express my gratitude for his availability for consultation on the challenging parts of the project and for offering his vast experience and knowledge in the subject matter to me.

I am also thankful to my examiner, Dr. Peter Akuon, for taking the time to go through this documentation and also listening to the presentation of the project, the time and effort spent is well appreciated.

Above all I would like to thank God for giving me the energy and ability to work on this project and produce something.

Contents

List of Figures

List of Tables

List of Abbreviations

RF - Radio Frequency
Kbps - Kilobytes per second
Mbps - Megabytes per second
Gbps - Gigabytes per second
IoT - Internet of Things
MITM - Man In The Middle
eNB - Evolved Node B
LTE - Long Term Evolution
LAN - Local Area Network
Wi-Fi - Wireless Fidelity
GHz - GigaHertz
TTP - Trusted Third Party
UE - User Equipment
RAN - Radio Access Network
CN - Core Network
IP - Internet Protocol
NMT - Nordic Mobile Network
AMPS - Advanced Mobile Phone Services
TACS - Total Access Communication Systems
MIN - Mobile Identification Number
ESN - Electronic Serial Number
GSM - Global System for Mobile Communication
GPRS - General Packet Radio Services
EDGE - Enhanced Data rate for GSM Evolution
MS - Mobile Station
BS - Base Station
BSS - Base Station Subsystem
BSC - Base Station Controller
NSS - Network Subsystem
ME - Mobile Equipment
SIM - Subscriber Identity Module
IMSI - International Mobile Subscriber Identity
MSC - Master Switching Centre
HLR - Home Location Register
VLR - Visitor Location Register
AuC - Authentication Center
TMSI - Temporary Mobile Subscriber Identity
SRES - Signed Result
SGGN - Serving GPRS Gateway Node
UMTS - Universal Mobile Telecommunication Subsystem

CS - Circuit Switched
PS - Packet Switched
SGSN - Serving GPRS Support Node
GGSN - Gateway GPRS Support Node
UTRAN - UMTS Terrestrial Radio Access Network
AV - Authentication Vector
RAND - Random Challenge
XRES - Expected Response
IK - Integrity Key
AUTN - Authentication Token
MME - Mobile Management Entity
HSS - Home Subscriber Server
EPS - Evolved Packet System
AKA - Authentication Key Agreement
RRC - Radio Resource Control
NIST - National Institute of Standards and Technology
SSP - Secure Simple Pairing
DH - Diffie Hellman
ECDH - Elliptic Curve Diffie Hellman
IEEE - Institute of Electrical and Electronics Engineers
WEP - Wireless Equivalent Privacy
WPA - Wi-Fi Protected Area
IV - Initialization Vector
MIC - Message Integrity Code
TK - Temporary Key
KA - Key Agreement
KDF - Key Derivation Function
KE - Key Encapsulation
KW - Key Wrapping
QKD - Quantum Key Distribution
AES - Advanced Encryption Standard
RSA - Rivest-Shamir-Adleman
ASCII - American Standard Code for Information Interchange

Abstract

This project proposes the implementation of a framework for secure communication that can be used in the implementation of Local Area Networks and various IoT systems. It also seeks to create an educational platform that may be built upon by local researchers and students. The implementation uses an nRF24L01 radio module designed by Nordic Semiconductors and also creates a few protocols to govern the communication. The protocols are implemented on top of the existing nRF protocols.

The report does a small analysis of various encryption algorithms and implements RSA and Diffie-Hellman. A brief analysis of the resultant system is also performed. The implemented system may easily be applied to Local Area Networks for communication between users in a given organisation. Using it may save on costs as compared to other possible methods of implementing.

Chapter 1: Introduction

1.1 General Background

Cellular technology keeps evolving as new innovations are discovered and a demand for higher communication quality keeps growing. In the 1980s 1G technology was released to the public. 1G supported voice data only. It had poor security, the voice quality was fairly low and it was prone to dropped calls. Its maximum speed was 2.4Kbps. At the time though 1G technology was the best available for long distance communication. Currently the world is transitioning into 5G. The standards that govern 5G technology dictate that 5G is to deliver astronomical speeds of up to 20Gbps. This will enable the development and growth of many industries such as IoT(Internet of Things).

As the quality of communication improves, the world is becoming more and more dependent on this technology. With 1G, when only voice communication was possible, the network was susceptible to various forms of attack such as MITM (Man in the middle attack), eavesdropping, forgery to name but a few. This however was not as costly since the confidentiality was not as important. However as the quality of communication continued to improve the integrity of the data became more and more critical. Nowadays, critical data is transmitted through cellular technology. As IoT continues to develop, the integrity of communication channels becomes more important. From banking details to health records, it is of utmost importance that this information does not get into the wrong hands. Thus the poor security that was used 40 years ago simply will not work.

Apart from improved speed for these networks, these standards also defined the minimum security requirement for these networks. For cellular networks, the security mechanisms can be grouped in two:

- The first set contains all the so-called network access security mechanisms. These are the security features that provide users with secure access to services through the device (typically a phone) and protect against attacks on the air interface between the device and the radio node (eNB in LTE and gNB in 5G).

- The second set contains the so-called network domain security mechanisms. This includes the features that enable nodes to securely exchange signaling data and user data for example between radio nodes and core network nodes

While cellular technology continues to grow, it is always accompanied by the additional overhead of needing the mobile service provider to provide the connectivity. This has the added cost that the transmission power needs to be very high which increases power consumption. Thus, for smaller networks such as LAN (Local Area Networks) e.g. to govern communication within a building, and IoT communication, it is not optimal to use cellular technology, in this case, implementations have opted for different approaches such as Wi-Fi and bluetooth.

1.2 Problem Statement

As IoT continues to develop, it is essential that there are multiple options to provide communication access between the devices. Standards need to be set for the different communication platforms so as to provide variety for the different implementations. Power consumption is also an issue that needs to be addressed as it is not feasible to rely on long range communication for shorter ranges. While there exists communication structures for Wi-Fi and bluetooth, an additional framework will improve the diversity while also providing a local implementation that can be used as an educational tool.

1.3 Objectives

- To implement a small scale working model of cellular transmission station together with user equipment that work on the Industrial, Scientific, and Medical radio bands (2.4 GHz).
- To implement different security measures.
- To analyse time taken for implementation of the various security algorithms.

1.4 Justification

Currently in the country already designed and tested algorithms are implemented by the major mobile network providers. This means that when an exploitation of the existing infrastructure happens we are forced to wait for solutions from outside. It also means that there is a lot less privacy in the networks. Because of this mobile money has suffered a lot of attacks from ill-motivated parties which has led to a lot of losses. Having a mobile system that can be used for testing and design can enable the security enforcers to be one step ahead of the criminals which is likely to reduce the amount of exploitation.

Having a small scale model will also enable the possibility of having local area mobile network communication that can be used to implement LAN(Local Area Networks) without the requirement of a lot of cabling and infrastructure layout.

1.5 Scope of the Project

The project was intended to implement a small scale version of a mobile transmission station. The project focuses on security and breach-ability of the network and not on the range of coverage and quality of the network.

Chapter 2: Literature Review

2.1 Modeling Threats to Mobile Networks

Before embarking on the security development it is important to define terms so as to remain within scope. For this project, we shall use the cryptographers definition of security. This is that security is the art of sharing secrets[1].

2.2 Models of Security Systems

2.2.1 Minimalistic model

A simple model for analysing system security is presented in [1]. In this case the authors gave the following mental aid to aid in understanding the network security. In this model the communication

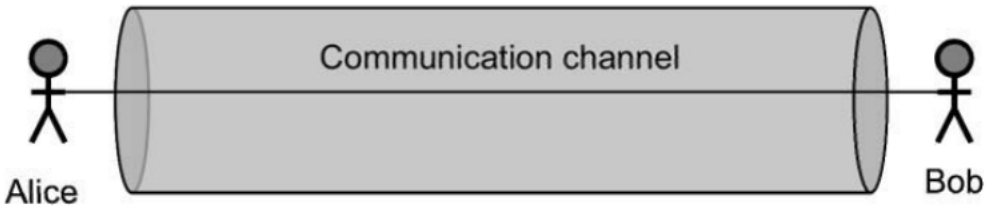

Figure 2.1: A minimalistic communication model

system is considered to have two participants, Alice and Bob, and a communication channel. The participants are considered to trust each other. While this model is often used in cryptography systems it is not sufficient for communication systems as it hides a lot of vulnerabilities that might be present.

2.2.2 Model with Communication Channel

The authors of [1] suggested a slightly more detailed model for analysing security systems. In this model the communication channel is shown to have at least one telecommunication infrastructure and a third party who is in charge of this infrastructure. For analysis purposes, one of the following is usually assumed:

- There is trust between Alice and Bob and the communication channels capacity to provide the necessary services(Private Network).
- There is trust between Alice and Bob, but they do not trust the crossed infrastructure (Public Network).
- Alice and Bob trust the communication channel but do not trust each other. They thus require a trusted Third party(TTP), the communication channel, to provide the necessary trust.

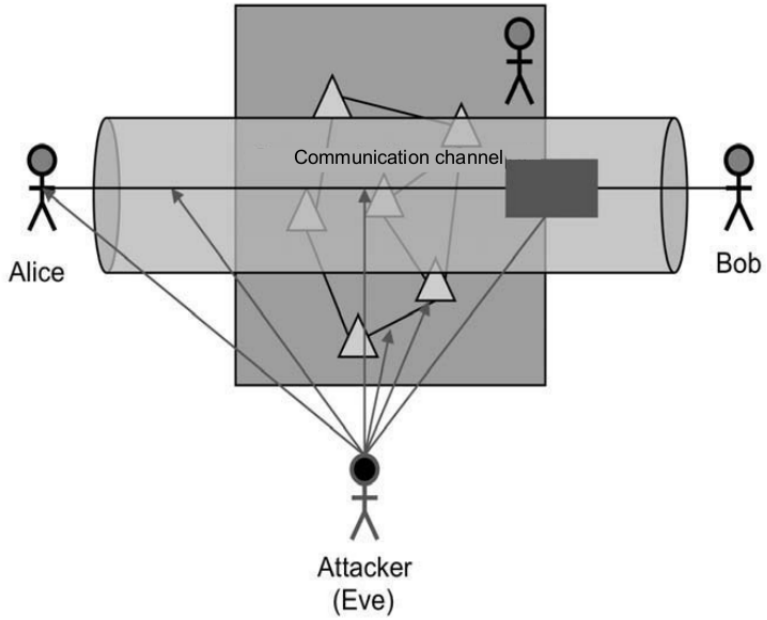

Figure 2.2: Model with communication channel and attacker

The attacker Eve on the other hand has the following options of attack if they want to interfere with the communication between Alice and Bob. They could:

- Eve could attack one of the communicating channels, for example through malicious code.
- Eve could attack the communication channel linking one of the users to the communication channel.
- Eve could attack the telecommunication system itself, for example Eve may try to masquerade as a legitimate part of the infrastructure.
- Eve could convince the communication channel that she is one of the genuine communicating parties.

2.2.3 Bhadra Framework

The authors of [2] saw it fit to have a more dedicated model for analysing mobile security. They first considered the topology of modern mobile communication systems.

The topology as seen in Figure 3 can be thought of as containing 6 sections for consideration:

1. The User Equipment(UE): This is the mobile device used by the subscriber.
2. Radio Access Network(RAN): This is the first point of Network Access.
3. The Core Network(CN): Manages mobility of usersbu interacting with other network operators and delivers telephony services.
4. Service Application Network: Contains Billing and Charging services, IP and multimedia services and other value added services.
5. Operation Support and Maintenance Network

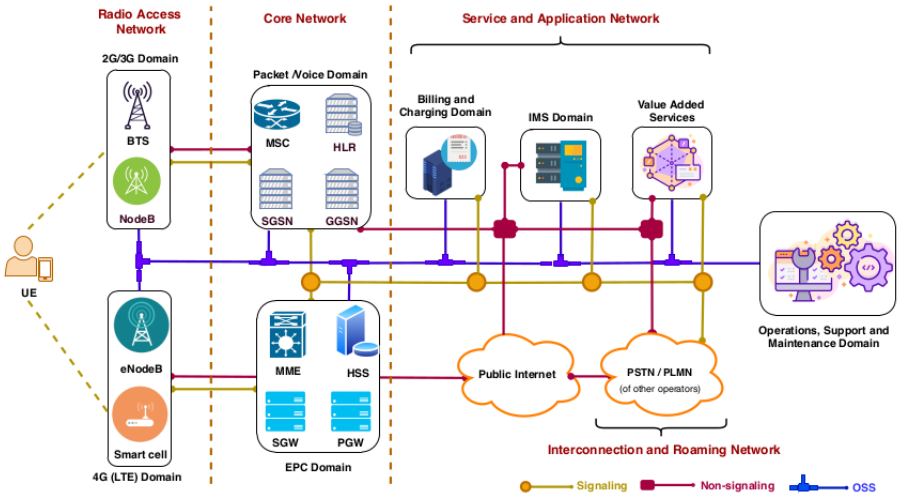

Figure 2.3: Overview of mobile network topology

6. Interconnection and roaming network.

The Bhadra framework then categorises the threats to the network based off of the specific sections of the network that are targeted. The attacks are farther split into three phases:

1. Attack Mounting Phase: This is where the adversary finds a weak point as its target to mount the attack and ensures that its control persists as long as it is required.
2. Attack Execution Phase: the adversary launches the attack, to achieve its main objectives, based on the preparation and information that it has sought from the previous phase.
3. Result Gathering Phase: theadversary achieves his main objectives.

2.3 Overview of Security Features in Different Mobile generations

For this analysis we shall consider Authentication of Users and the networks and also consider encryption used [3],[4]

2.3.1 1G

There were 3 different independent systems for 1G technology. Namely, Nordic Mobile Telephony(NMT, used in parts of Europe and USA), Advanced Mobile Phone Services(AMPS) and Total Access Communication system (TACS) AMPS and TACS lacked authentication and data encryption services. This made it easy for attackers to extract Mobile Identification Numbers (MIN) and Electronic Serial Number (ESN). It was thus easy for attackers to impersonate users. NMT employed voice scrambling at the mobile phone and base station [5].

2.3.2 2G

We shall consider 2G also known as GSM (Global System for Mobile Communications), 2.5G (General Packet Radio Services) and Enhanced Data rates for GSM Evolution (EDGE). In GSM 2 security measures were taken, authentication and encryption. In order to understand the security measures we must first review the architecture. The GSM architecture is divided into 3 subsystems: The mobile station(MS), the Base Station Subsystem(BSS) and Network Subsystem (NSS) as shown in figure 3. The mobile station consists of Mobile Equipment(ME) and Subscriber Identity Module (SIM) which contain secret information that is essential for the security (the International Mobile Subscriber Identity Module IMSI and secret Key Ki). The BSS consists of the Base Station Controller(BSC) and Base Station (BS). The BSC is connected to Mobile services Switching Center (MSC) in the third part of the network. The MSC also contains several databases such as the Home Location Register (HLR) and Visitor Location Register (VLR). The MSC, incooperation with HLR and VLR, provides numerous functions including registration, authentication, location updating, handovers and call routing.

In the entire 2G authentication process, the 3 main actors are, MS, MSC and HLR/AuC (Authentication Centre). The AuC contains an individual 128 bit key (K_i), which is a copy of K_i on the SIM. The authentication procedure is as follows:

1. The mobile station sends its Temporary Mobile Subscriber Identity (TMSI) to VLR in its request for authentication.(IMSI is only used when the phone is switched on for the first time TMSI is used on subsequent times).
2. IMSI is then obtained from the old VLR using TMSI and then sent to the corresponding AuC/HLR.
3. The AuC uses authentication Algorithm A3 and ciphering key generation algorithm A8 to create encryption key (K_c) and signed result (SRES) respectively.
4. HLR sends K_c, RAND and SRES to VLR.
5. VLR sends RAND challenge to MS and asks it to generate SRES and send it back.
6. MS creates K_c and SRES uses A3 and A8 algorithms and the inputs K_i and RAND chanllenge
7. MS stores K_c to use for encryption and sends back SRES to VLR.
8. VLR compares SRES from HLR and MS. If they match Authentication was successful otherwise Authentication has failed

A flow chart of the process can be seen below. [3, 6]

During encryption the A5 algorithm is used. It utilises the K_c generated during the authentication process and a frame counter F_n to generate a Pseudo random(PRAND) 228 bit key that is then XOR'd with them with the data to be transmitted. As the XOR operation is not very difficult to reverse, A5's security relies on the generation of the PRAND key. There are 3 versions of A5. A5/0 the Unencrypted Stream Cipher offers the weakest encryption. It's output is just the input plain text. A5/1 which offers strong encryption and is used in Europe and USA. A5/2 which was shown to have many back door. Even the strong A5/1 was shown to have several weaknesses and vulnerabilities[7].

GPRS improved security measures by ensuring data remains encrypted on all radio links. Cipher-

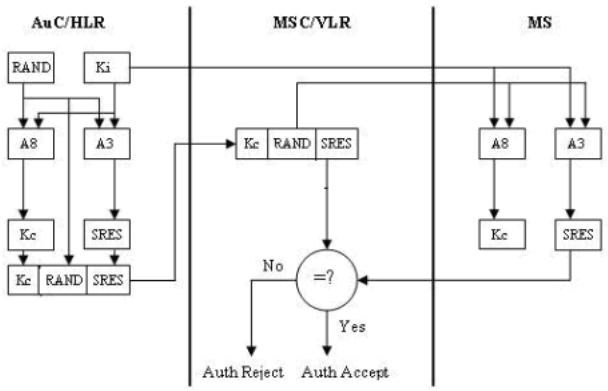

Figure 2.4: GSM authentication flowchart

ing was terminated at the Serving GPRS Gateway Node (SGGN) instead of at the BTS.

2.3.3 3G

Universal Mobile Telecommunication System(UMTS) was launched as an improvement to GSM. In order to understand the security model of UMTS it is important to note, GSM used Circuit Switched(CS) technique for its voice communication while GPRS used Packet Switched (PS) technique through the use of some extra nodes namely Serving GPRS Support Node (SGSN) and Gateway GPRS Support Node (GGSN). In order for UMTS to provide backward compatibility it required to provide both PS and CS services. It did this by incorporating GPRS nodes together with UMTS Terrestrial Radio Access Network (UTRAN).

Similar to GSM and GPRS UMTS employed the concept of the Authentication Vector AV(the triple consisting of K_c, RAND, SRES) except instead of having 3 components, the AV in UMTS had 5 components namely [6]:

- Random Challenge (RAND)
- Expected Response (XRES)
- Key for Encryption (CK)
- Integrity Key (IK)
- Authentication Token (AUTN)

The Authentication process is as follows:

1. The UE device tries to register to the network by initiating location and routing area update procedures.
2. The UE transmits IMSI to the nodeB (UMTS BS).
3. The MSC uses IMSI to retrieve authentication information for the UE.
4. The network returns a 64-bit message authentication code (MAC) that consists of K_i, a 48-bit sequence number(SQN), 128-bit RAND and 16 bit authentication management field (AMF)that allows to trigger session key changes or cryptographic algorithm changes.

5. K_c and I_k are generated. Also to avoid SQN based attacks Anonymity Key (AK) is also generated
6. The 5 element Authentication vector discussed above is sent to the UE.
7. UE uses AUTN to verify the procedure was initiated by an authorized network.
8. UE then sends a 32 to 128 bit response (RES) that consists of K_i and RAND back to nodeB
9. Authentication is successful if $RES = XRES$

As can be seen in 3G authentication is done both by the UE and the BS [3]. For encryption in 3G several methods were employed making it more difficult for attackers to figure it out for all the different networks. Some of the popular ones include, KASUMI cipher, SNOW3G and Rijndael cipher.

2.3.4 4G

In Long Time Evolution(LTE/4G) communication between serving networks and home networks is based on IP. A typical 4G network consists of the Evolved NodeB (enodeB) the successor to the 3G nodeB, Mobility Management Entity(MME) which is responsible for authentication, Home Subscriber Server (HSS similar to HLR in GSM and UMTS). The core entities that are connected over an IP network are collectively referred to as Evolved Packet System (EPS). The 4G- EPS AKA (Authentication and Key Agreement) is as follows:

1. The UE completes the Radio Resource Control Procedure (RRC) and sends an Attach Request to the MME.
2. The MME sends an Authentication Request, including UE identity (i.e., IMSI) and the serving network identifier, to the HSS located in the home network.
3. The HSS performs cryptographic operations based on the shared secret key K_i to derive one or more AV's similar to those in 3G.
4. The HSS sends the AV to the MME in an Authentication Response Message. The AV consists of AUTH token and expected authentication token XAUTH
5. The MME sends an authentication request to the UE which includes the AUTH token.
6. UE validates AUTH token based off the token it produced with K_i. If the authentication succeeds UE considers network to be legitimate and sends an Authentication Response message back to MME including RES token
7. The MME compares the RES token with an expected response (XRES) token. If they are equal, the MME performs key derivation and sends a Security Mode Command message to the UE, which then derives the corresponding keys for protecting subsequent NAS signaling messages.
8. After the UE also derives the corresponding keys, subsequent communication between the UE and the eNodeB is then protected.

The flow chart below summarises the 4G authentication procedure[8, 9].

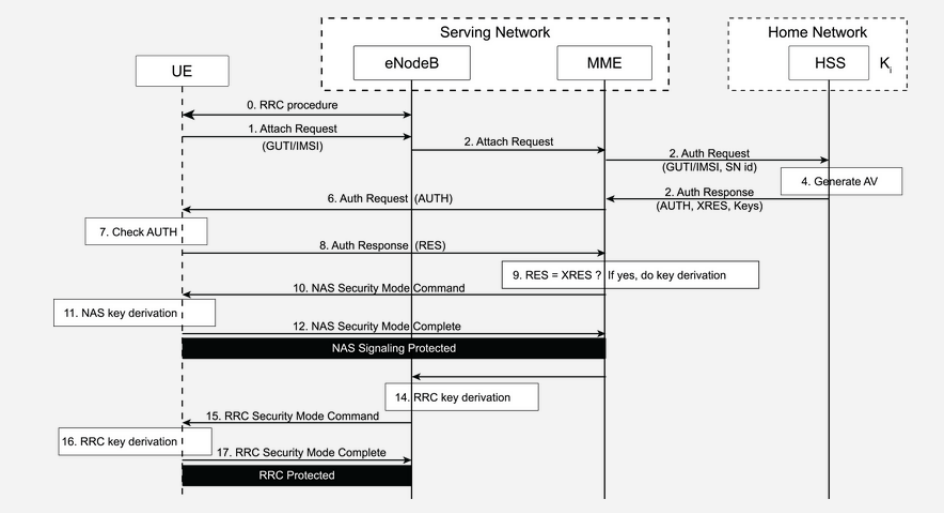

Figure 2.5: Flow chart explaining the 4G authentication Procedure

2.3.5 Bluetooth

The bluetooth standard specifies five security services, namely:

1. Authentication
2. Confidentiality
3. Authorization

The bluetooth specifications then defines 4 modes within which every bluetooth device must operate [14]:

1. Security Mode 1: In this mode, security functionality such as encryption and authentication are never initiated. Due to this, the NIST recommendation is to never use security mode 1.
2. Security Mode 2: Here security procedures are initiated after link establishment but before logical channel establishment. In this mode, there exists a local security manager who controls access to specific services. This is achieved by varying the security policies and trust levels to restrict access. In this mode, the notion of authorization is introduced.
3. Security Mode 3: Link level security is enforced in this level. This means that security is initiated before the physical link is fully established. Both authentication and encryption are mandated.
4. Security Mode 4: Similar to mode 2, in mode 4 security procedures are initiated after physical and logical link is setup. This mode uses Secure Simple Pairing (SSP), in which Elliptic Curve Diffie-Hellman (ECDH) key agreement is used.

2.3.6 WI-FI

The IEEE standard 802.11 based wireless connection defines the standards to be used for security in Wi-Fi links [15]. There are 2 main protocols:

- Wired Equivalent Privacy (WEP)
- Wi-Fi Portected Access(WPA).

WEP uses the RC4 [16] encryption scheme and CRC-32 for data integrity. The encryption process used in WEP is summarised in the flowchart below: IV represents the initialisation vector that

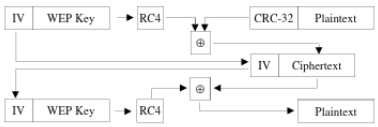

Figure 2.6: WEP Encryption Process

is 3 bytes long. WPA is an improvement of WEP. It addresses the problem of statistical attacks by using random key generation.It also has additional properties such as Message Integrity Code (MIC) and Key hash function to avoid IV attacks. The figure below shows the operation of the WPA Encapsulation process. TK, DA and SA denote respectively teh temporary key, the sender and destination addresses. || is the concatenation operator.

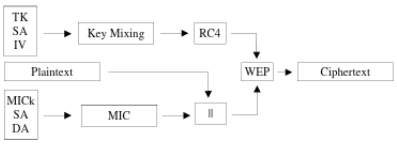

Figure 2.7: WPA encapsulation process

2.4 Analysis of Encryption Algorithms

Encryption algorithms can be classified into two categories, symmetric key encryption algorithms and asymmetric key encryption algorithms.

Symmetric systems are those in which the keys (values used for encryption and decryption) at the transmitter and receiver either are the same or can be easily computed from each other [17] . One of the main challenges in symmetric key encryption is sharing of keys between parties. The strategies used can be

1. Key Agreement (KA): the key is established using a Key Derivation Function (KDF) such as Diffie-Hellman (DH) or Elliptic Curve Diffie Hellman(ECDH)
2. Key Encapsulation (KE) or Key Wrapping(KW): the key is encrypted and sent using asymmetric encryption algorithm.
3. Quantum Key Distribution (QKD): the key is shared using an optically linked connection that cannot be eavesdropped.
4. Out-of-band Procedures: This is where the key is shared using for instance, a previous telephone call, or during a previous physical encounter.

5. Key Sharing: this is whereby a key is created using parts held by different persons.

Some of the common Symmetric key encryption algorithms include [18]:

- AES
- Caesar cipher
- Playfair cipher
- Modular Multiplication Block Cipher
- RC4

In asymmetric attacks, the encryption and decryption keys cannot be easily obtained from each other. There are three possibilities:

- Forward asymmetric: The decryption key cannot be easily computed from the encryption key.
- Backward asymmetric: The encryption key cannot be easily computed given the decryption key.
- Bidirectional asymmetric: Neither the encryption nor decryption keys can be easily computed from each other.

In asymmetric encryption, there is no need for key exchanges to be made. In this case one of the keys is made public while the other is made private. Asymmetric systems are usually based off of problems known to be exceedingly complex such as the knapsack problem or the factorization problem [19] [20] [21]. Below an asymmetric algorithm that relies on the difficulty of integer factorization (RSA) and a key exchange algorithm that also relies on the discrete logarithm problem [22] are analysed below.

2.4.1 RSA

In order to understand the working of RSA algorithm, we must consider Fermat's Little theorem: It states that if M is relatively prime to n, then

$$M^{\phi(n)} \equiv 1(modn). \tag{2.1}$$

where $\phi(n)$ is Euler's totient function. Initially,the authentication node (the "base station"),selects 2 large prime numbers r and s. It is important that they are large because if they are small a brute force attack may be used to break the encryption. A number e that is co-prime with the product $(r-1)(s-1)$ is chosen. The reason for the co-prime condition shall be made apparent in the proof of correctness below. The public key that shall be made known to all end stations shall consist of the pair of integers e and n. The multiplicative inverse of e modulo $(r-1)(s-1)$ which we shall denote by d is the private key and is kept secretly in the base station. Note if d is the modulo inverse of e mod $(r-1)(s-1)$ it implies that

$$de \equiv 1mod((r-1)(s-1)) \tag{2.2}$$

In order to obtain the private key d an attacker would need to be able to obtain either r or s from n but if both r and s are chosen to be large numbers, this would take too long and would not be possible.

Let M be the message to be encrypted. The encryption procedure involves raising M to e i.e. obtaining M^e modulo n. For non-numerical data, this is done by first converting it to numerical format e.g using ASCII codes then splitting it into small blocks and raising the smaller blocks individually. An efficient algorithm for doing this is presented below.

1. Let e_k ,e_{k-1} $\cdots$ e_1, e_0 be the binary representation of e (e_0 is least significant bit and e_k is most signficant.
2. Set the variable C to 1.
3. Repeat steps 3a and 3b for i = 0,1, $\cdots$,e_{k-1}, e_k:
   (a) If ei = 1, then set C to the remainder of $C \cdot M$ when divided by n
   (b) Set M to the remainder of M^2 when divided by n.
4. Halt. Now C is the encrypted form of M .

The decryption is done in a similar way and involves raising the encrypted message C to d i.e. obtaining C^d modulo n.

The proof of correctness relies of the following properties of Euler's totient function: For all prime numbers p,

$$\phi(p) = p - 1. \tag{2.3}$$

If $x = a \cdot b$ where a and b are prime then

$$\phi(x) = \phi(a) \cdot \phi(b) \tag{2.4}$$

From the above properties we get

$$\begin{aligned} \phi(n) &= \phi(r) \cdot \phi(s) \\ \phi(n) &= (r-1) \cdot (s-1). \end{aligned} \tag{2.5}$$

The deciphered D text which can be written as

$$D \equiv C^d \ mod \ (n) \tag{2.6}$$

but

$$C \equiv M^e mod(n) \tag{2.7}$$

thus

$$\begin{aligned} D &\equiv (M^e)^d \ mod \ (n) \\ &\equiv (M^{ed} \ mod \ (n)) \end{aligned} \tag{2.8}$$

But by design e and d are chosen so that ed is congruent to 1 mod (r - 1)(s - 1). D can thus be rewritten as

$$\begin{aligned} D &= M^{1+k(r-1)(s-1)} \\ D &= M\left(M^{k(r-1)(s-1)}\right) \end{aligned} \tag{2.9}$$

but if M is co-prime with n, then from Fermat's little theorem we will have:

$$D \equiv M(1)^k \ mod \ n \tag{2.10}$$

which is just the original message. Thus RSA is capable of encrypting and decrypting messages to be sent.

2.4.2 Diffie-Hellman Key Exchange

Another technique that is used to achieve encryption is the Diffie-Hellman Key exchange. The figure below shows the flow of information in a conventional cryptographic system. In this model,

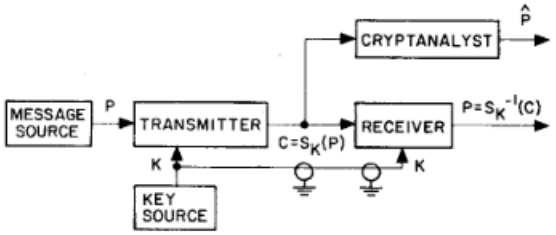

Figure 2.8: Flow of information in a conventional cryptographic system

the message to be transmitted, P, is to be communicated over an insecure channel. The transmitter generates $C = S_k(P)$ and transmits it instead. The Encryption Key K thus needs to be transmitted over a secure channel [12]. When the receiver receives the message C the apply the inverse operation $S_k^{-1}(C)$ which is $S_k^{-1}(S_k(P))$ which is P. The Diffie-Hellman Key exchange proposes a method to transmit the key K over the unsecure channel. It relies on the difficulty of computing logarithms over a finite field $GF(q)$ with a prime number q. Let

$$Y = \alpha^X \ mod \ q \qquad for \ 1 \leq X \leq q-1 \tag{2.11}$$

where α is a fixed element. Y is the logarithm of X to the base of α mod q. Calculating Y from X is easy and can be done with the algorithm shown in RSA. However, computing X from Y is difficult and would require $O(\sqrt{q})$ operations.

This is thus the principle employed by the Diffie-Hellman key exchange. The procedure is as follows:

1. Each user generates an independent random number X_i chosen uniformly from the set of integers [1,...,q-1].
2. Each user privately stores X_i but may publicly share $Y_i = \alpha^{Xi} \ mod \ q$
3. When 2 users are communicating they use $K_{ij} = \alpha^{XiXj} \ mod \ q$

User i obtains K_{ij} by obtaining the publicly shared Y_j and raising it to X_i i.e.

$$K_{ij} = \alpha^{X_iX_j} \ mod \ q = Y_j^{X_i} \ mod \ q \tag{2.12}$$

Thus, Diffie Hellman can be used to exchange private keys over a public channel. Hence enabling end to end encryption.

Chapter 3: Design

3.1 Hardware Selection

The components required are:

1. A communication module that will be used for both the user equipment and base station
2. Micro controller to run the user equipment.
3. Input and output for the user equipment.
4. A computer to act as a base station.

3.1.1 End Equipment Design

The micro-controller chosen to run the system was an Atmega32.It was preferred due to its larger size and number of pins(40). It was programmed using C programming language. A 4 x 4 membrane keypad was chosen to take the user input and a 16 x 2 LCD display with PCF8574 I2C module to display the output to the user. These were chosen due to their ease of use and number of pins required. Images of the equipmment used are shown below. The voltage requirements of

Figure 3.1: Atmega32

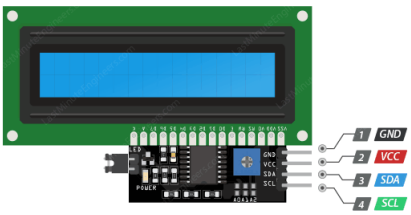

Figure 3.2: 16 x 2 LCD with PCF8574 I2C module

the different components are as follows:

1. Atmega32 - 5V
2. LCD - 5V
3. NRF radio module - 3.3V

Figure 3.3: 4x4 membrane keypad

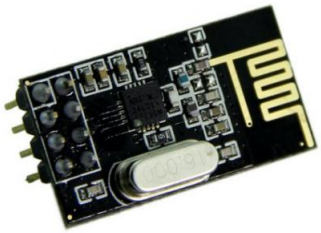

Figure 3.4: nRF24L01 radio module

It is noted that the NRF module has a different voltage requirement from the Atmega32 and the LCD display. Farther the battery that was to be used produces varying voltage of around 9V. In order to solve this, a buck converter together with a 3.3V linear regulator were used so as to supply the different devices. The circuit diagram below shows a schematic of how the components were

Figure 3.5: Adjustable Buck Converter

Figure 3.6: AMS1117 3.3V linear regulator

connected together.

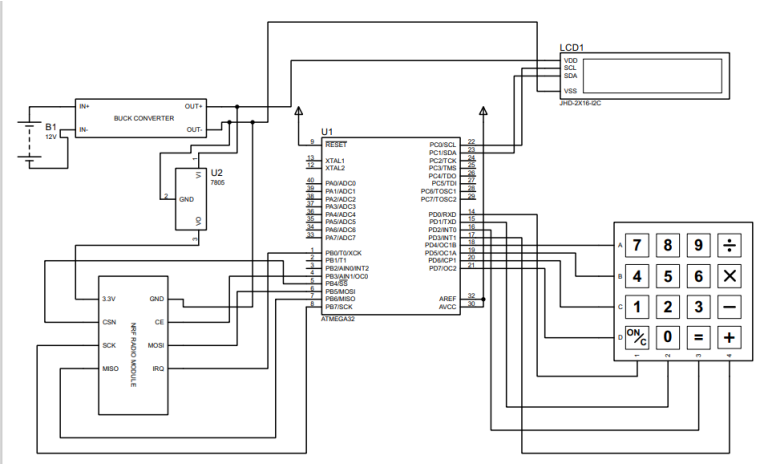

Figure 3.7: Circuit diagram for end equipment

3.1.2 Base Station Design

For the base station, a simple Arduino Uno with an NRF radio Module connected to its SPI bus was used. It was then connected to a laptop using a USB type B connector. On the laptop, a serial monitor was used to read the transmitted packets and monitor the operation of the base station.

3.2 Topology of System

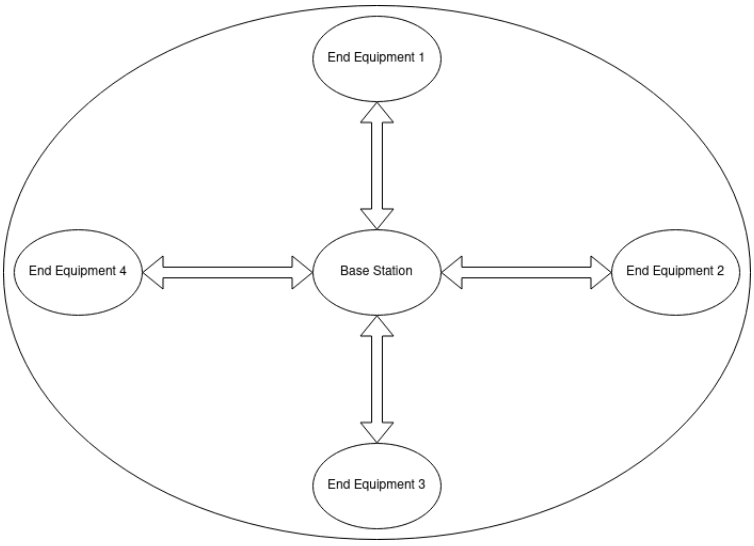

Figure 3.8: Design of a single cell

The complete network may be divided into smaller cells as shown above. As the NRF allows for multiplexing of up to 5 devices, Each base station can serve a maximum of 5 devices. This can however be improved by farther software multiplexing. Communication between the components is outline below.

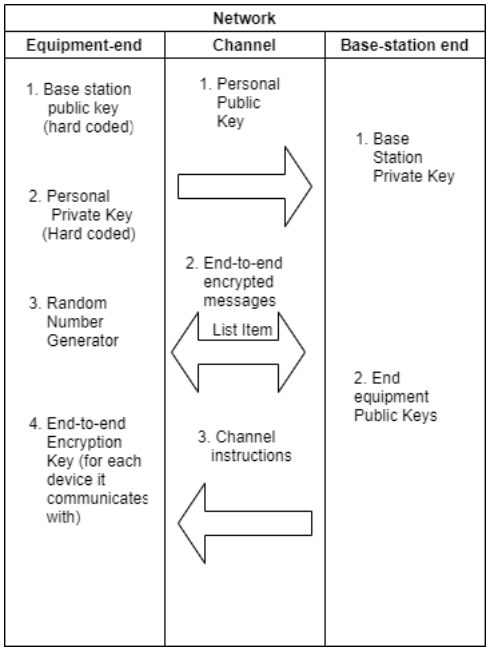

Figure 3.9: Topology of the system

The authentication and encryption data that shall be held by end user equipment include:

1. The base station Public Key. This will be hard coded in every item so as to reduce packets sent in the channel.
2. Personal private key. These are initialized in every equipment. Packets from the base station are encrypted using the accompanying public key. The personal private key is never to be sent through the channel and should not be known by anybody other than the equipment itself.
3. Random Number Generator. To be used to generate keys for end to end encryption. The results should not be shared through the channel.
4. End-to-end Encryption Key. For each pair of device that communicates with each other, an end-to-end encryption key is required. It is generated per session and should not be shared in the channel.

As can be seen, a lot of encryption data is held in the end equipment never to be shared through the channel. Data that is actually shared in the channel includes:

1. End-equipment public key. It is sent from the end-equipment to the base station when an equipment joins the network. It is shared once during the initialization stage.
2. End-to-end encrypted messages. These are messages that are to be sent from end-equipment to end-equipment. They pass through the base station as the end equipment might not be in range with each other. They are end-to-end encrypted messages and thus cannot be deciphered by the base-station.
3. Channel instructions. These are also sent during the initialization stage. They include the channels that the user equipment should use when communicating with the base station.

Finally. The encryption data that is stored in the base station includes:

1. Base Station Private Key. This is the key that is used by the base station to decipher messages sent to it.
2. End equipment public keys. These are the keys that are sent to the base station during the initializing stage. Before the base station sends information to user equipment, it encrypts the data with these keys.

3.3 Packet Structure

The NRF24L01 radio module uses the Enhance ShockBurst Protocol which was discussed in chapter 2. In order to customise it for the system, we shall farther tailor it to capture more details that are relevant to the communication. The added information shall be added onto the protocol payload as it has a large enough size which can have a variable length so as to accommodate the additions.

Figure 3.10: Modified Payload Structure

The first parameter is the Packet type. For the design, 4 types shall be considered:

1. Initialization Packet
2. Channel Control Packet
3. End-to-end encryption initialization Packet
4. End-to-end encrypted Packet

The second parameter is the final intended final destination. This may either be a base station or another end equipment. It is 1 byte long as it only contains an ID.

The third parameter is the from Radio ID. It is 1 byte long as well. It contains the original ID of the sender.

The fourth and final parameter is the Packet Data. It can take any length from 1 byte to 20 bytes. Its length is determined by the Packet Type. The Initialization Packet packet Data is 8 bytes and contains the RSA keys n and d each 4 bytes long. The Channel Control Packet packet Data is 1 byte long and contains the channel the end equipment is to use when communicating. The End-to-End Encryption Initialization Packet is 4 bytes long since the only information that is sent is Y_i. The End-to-End Encrypted Packet is 20 bytes long. It contains packets that are intended for the end equipment and thus has the largest data size.

3.4 Prerequisite Computations

Before determination of the system, some computiations need to be done for computing the keys and encryption procedures. These techniques are outlined in the flowcharts below:

1. Computing modulo inverses for RSA encryption keys e and d: This is done using the Extended Euclid Algorithm. The algorithm takes as input 2 integers a and b and returns 3 integers (g,x,y) where g is gcd(a,b) and we have $a \cdot x + b \cdot y = g$. If the integers a and b are co-prime(i.e. g = 1), then x is known as the multiplicative inverse of a modulo b. If a is set to d in the RSA algorithm and b is set to n, then x will correspond to d thus all encryption keys will be known. The extended Eucledian algorithm has a complexity of $O(log(a))$

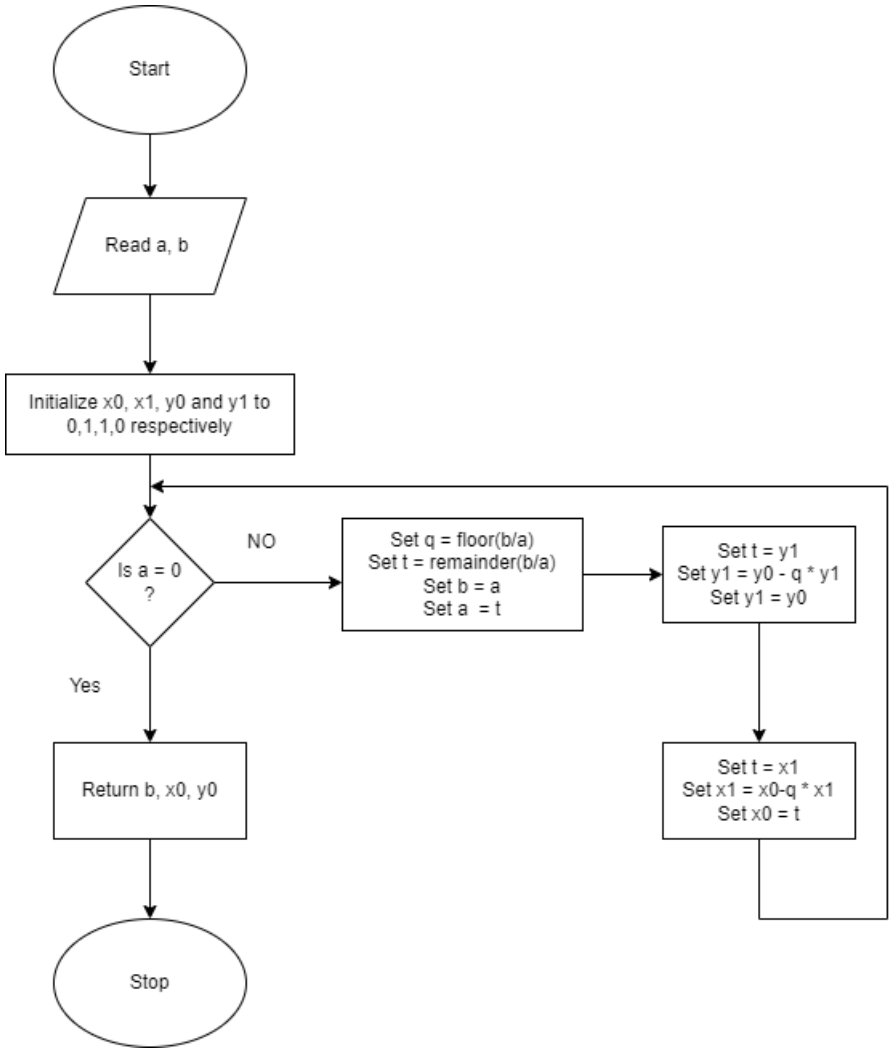

Figure 3.11: Flowchart for the process of computing modulo inverse

2. Computing Y_i from X_i for Diffie-Hellman Key exchange. The algorithm computes $g^{X_i} mod p$ as defined in chapter 2. This uses binary exponentiation thus has a complexity of $O(log(X_i))$

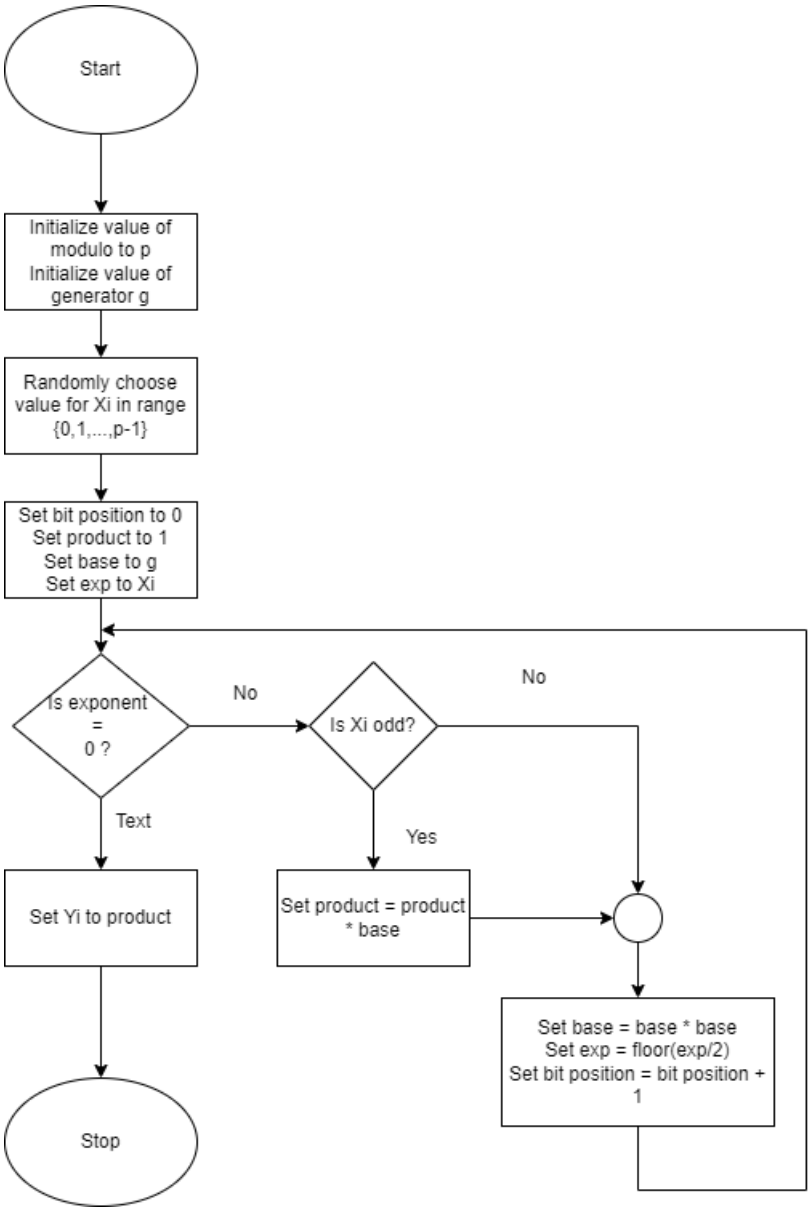

Figure 3.12: Computing Yi for Diffie Hellman Key Exchange

3.5 Authentication Procedure

The general procedure of initialization when an end equipment joins a network is shown in the flow-chart below. At the end of the initialization process, the end equipment's public key is stored in the base station registry and the end equipment can communicate with the base station successfully. In order for a malicious party to impersonate either the end equipment or the base station, they will require the private keys, either that of the end equipment or that of the base station. This is however not possible because these are never transmitted on the channel. The system is thus safe from impersonation attacks. Any external party may intercept the messages, however they

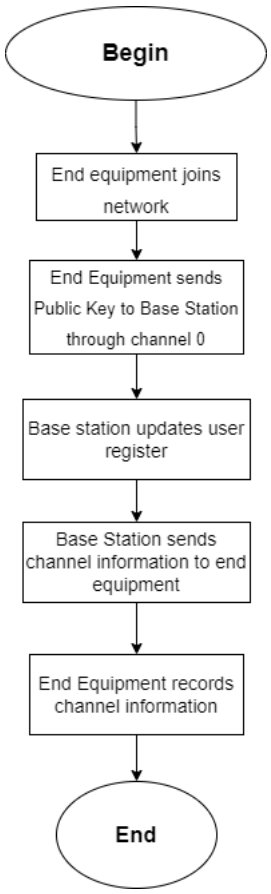

Figure 3.13: Flowchart of authentication procedure

will not be able to decipher the meaning from the message. The system is however still prone to denial of service Attacks (DoS). This can be done by a malicious party posing as a base station. The third party intercepts traffic intended for the base station and prevents them from getting to the base station. While the malicious user may not be able to get any meaning from the sent packets, they are able to prevent the end equipment from communicating with the base station.

The channel 0 is reserved for sending and receiving initialization information. For demonstration purposes, the keys for sending to the channel were:

3.6 Initialization Procedure

This is carried out at when the end equipment is powered up. Prior to initialization, some values have to be determined and set. These values include:

- Public Keys for RSA encryption for communication with base station.
- Public Keys and private Keys for RSA encryption when base station is communicating with the end equipment.
- generator g, modulus and X_i for Diffie-Hellman Encryption

With the above values already pre-computed, the initialization procedure is really short and is described in the flowchart below:

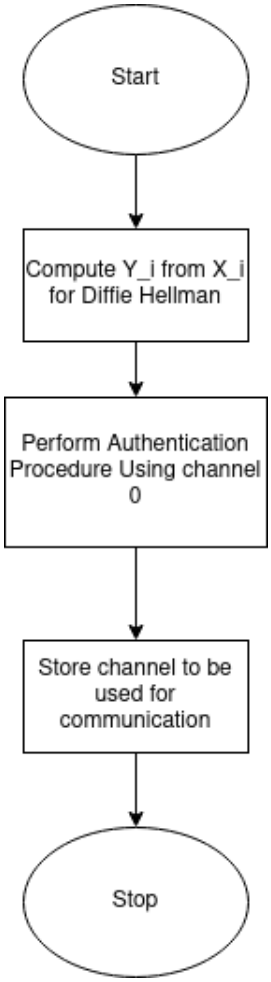

Figure 3.14: Initialization procedure

3.7 Encryption Procedure

Before packets are sent from the end equipment, they are encrypted at least once. However, in order to ensure that the base station does not have access to sensitive information, farther encryption is done using Diffie-Hellman algorithm. This ensures end-to-end encryption. The procedure below shows the procedure for end-to-end communication between equipment by the initialising device is as follows: Algorithm for initializing end to end encryption:

1. Repeat steps 2 and 3 till you receive the target receiver's Y
2. Send Y_mine
3. Wait 100ms
4. Set Y_theirs to the received value of Y
5. Compute the end-to-end encryption key as Y_theirs $\hat{X}$_mine
6. Compute modular inverse for decryption purposes
7. Encrypt message by using modular multiplication
8. Send Encrypted Message

The above algorithm is farther elaborated in the flow-chart below.

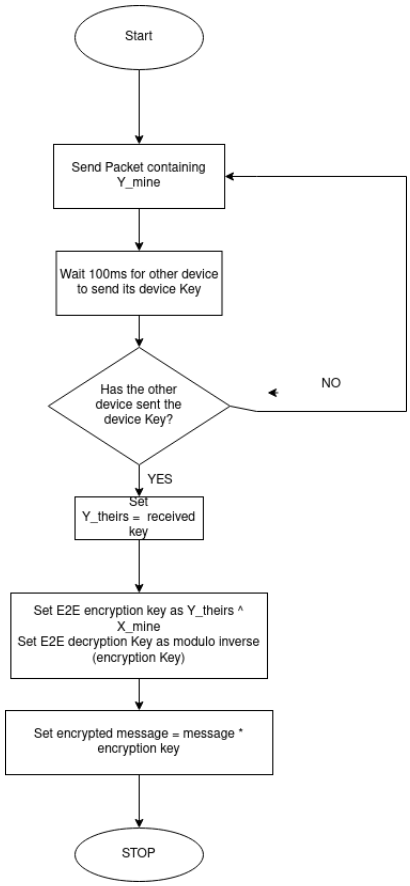

Figure 3.15: End to end encryption procedure by initiating device

3.8 Overall Design Of End Equipment Device

The overall functioning of the End Equipment Device can be summarised as follows:

1. Join the network by performing Authentication Procedure.
2. Repeat the steps 3 to 17 while the power is being supplied.
3. Check assigned channel for received communication.
4. If there is a received packet: perform steps 5 to 12
5. Decode the packet.
6. If the packet type is an End-to-End encryption initialization packet, perform steps 7 to 9
7. Obtain Y_theirs from the received packet
8. Send Y_mine to the device
9. Compute Diffie Hellman encryption key and decryption key
10. Else if the packet is an End-to-end encrypted packet perform steps 11 to 12
11. Decrypt the packet using the decryption key
12. Display the received message on the LCD
13. Check for input information using the keypad
14. If there is a message to be sent perform steps 15 to 17
15. If the end-to-end encryption key for the target device has not been initialized, perform end to end encryption initialization
16. Encrypt message
17. Send encrypted message.

The above algorithm is visually represented in the flowchart below.

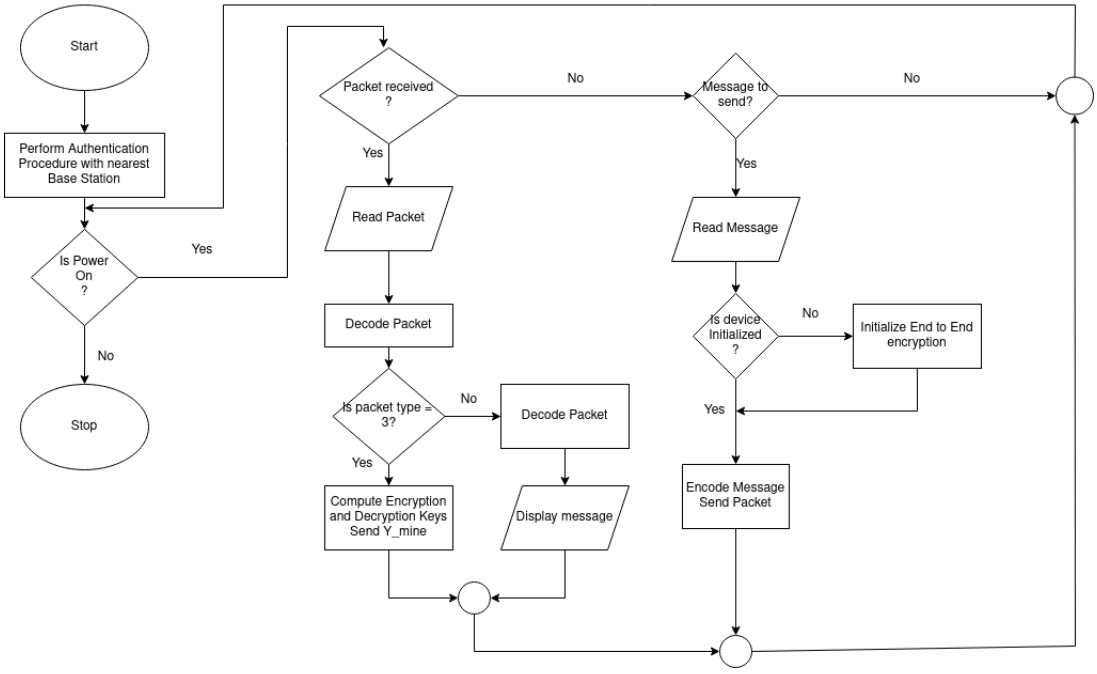

Figure 3.16: Overall design of end equipment programming

3.9 Overall Design of Base Station

The following algorithm summarises the operation of the base station.

1. Initialize variables.
2. Repeat steps 3 to 10 while the device is powered.
3. Check channel 0 for a device that wants to join the network.
4. If a device requesting to join the network, perform authentication of the device.
5. Iterate through all channels and check if there is a packet that was sent.
6. If there is a packet present do steps 7 to 10
7. Read packet
8. Decipher packet using private key
9. Encrypt packet using the public key of the final packet destination.
10. Send packet to final end device.

This is summarised in the flow-chart below.

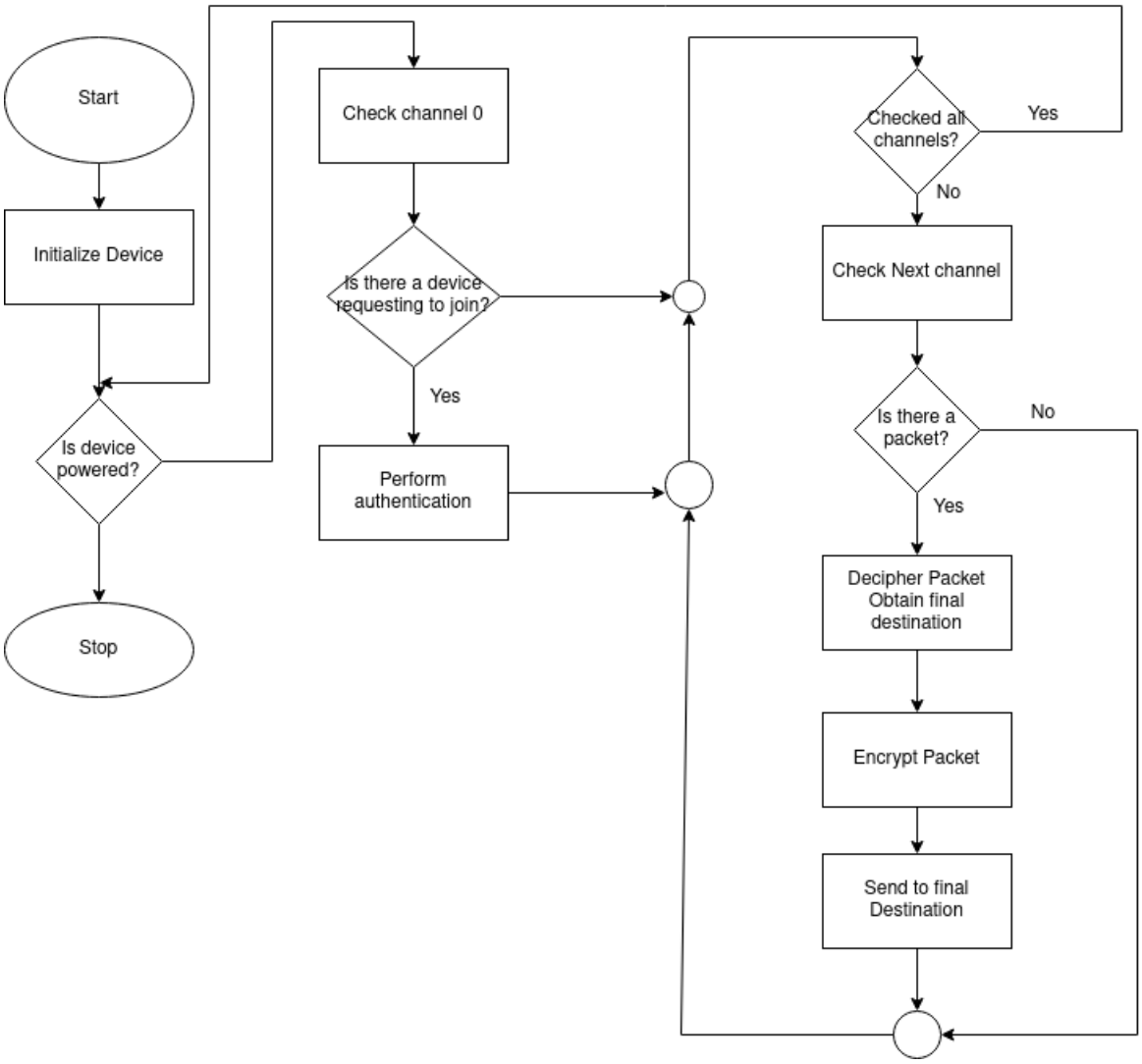

Figure 3.17: Overall design for Base Station

Chapter 4: Implementation

4.1 Software Implementation

4.1.1 Input Device

The 4x4 keypad consists of 8 pins, where 4 of them correspond to the rows of the matrix and the other 4 the columns of the matrix. When a button is pressed the corresponding row and column are connected. Thus if the column pins are set to inputs with internal pull-ups, the button pressed can be detected by setting the row value to ”LOW” and then reading the column values to see which column is ”LOW”. The code that performed this is shown in the figure below The key-mapping

```
while (row < 8){
        //Setting current row to  LOW
        PORTD &= ~(1 << row);
        //Setting column to 0 for each row that is set to low
        column = 0;
        //Checking state of each column
        while (column < 4){
                if ( (PIND & (1 << column)) == 0) {
                        key = KEYPAD_KEYS[row-4][column];
                        interpret_key(key);
                        updated = 1;
                }
                //Incrementing column
                column ++;
        }
        //Setting the row back to HIGH
        PORTD |= (1 << row);
        //Incrementing row
        row ++;
}
```

Figure 4.1: code to detect button pressed

design was based on the input devices of old cell phones where one key was used to represent multiple input characters (the ITUE standard - shown below). A slight adjustment was made due

Figure 4.2: Letter Mapping ITU-E standard

to the inclusion of the additional keys. The 4 additional keys were given the following functions The code snippet below shows the character order of the remaining 12 keys.

KEY	FUNCTION
A	Switch between different characters e.g. 2 to a
B	Delete last entered character (like backspace)
C	Change between typing message and typing the recipient's number
D	Used to send the message typed.

```
const char character_order[12][10] = {{'1','1','1','1','1','1','1','1','1','1'},
                                      {'2','a','b','c','A','B','C','2','2','2'},
                                      {'3','d','e','f','D','E','F','3','3','3'},
                                      {'4','g','h','i','G','H','I','4','4','4'},
                                      {'5','j','k','l','J','K','L','5','5','5'},
                                      {'6','m','n','o','M','N','O','6','6','6'},
                                      {'7','p','q','r','s','P','Q','R','S','7'},
                                      {'8','t','u','v','T','U','V','8','8','8'},
                                      {'9','w','x','y','z','W','X','Y','Z','9'},
                                      {'*','*','*','*','*','*','*','*','*','*'},
                                      {'0',' ','.',',','0','0','0','0','0','0'},
                                      {'#','#','#','#','#','#','#','#','#','#'}};
//order of characters pressed by the keypad
```

Figure 4.3: character order of remaining 12 keys

4.1.2 Output Device

A 16 x 2 LCD with PCF8574 I2C was used. Code to run it was taken from a project written by Alvyne Mwaniki [13]. With the two-wire interface(TWI also known as Inter-Intergrated Circuit) the LCD only used 2 pins. The functions lcd_print(), lcd_clear() and lcd_second_line(), were the functions that were used for the implementation. With the I2C module, writing onto the LCD simply involved sending commands to the module and then the module communicated with the LCD. Below is an example of how the lcd_second_line() function was implemented:

```
static inline void lcd_second_line(){
        lcd_command(LCD_CUR_LINE2);
}
```

Figure 4.4: Code snippet demonstrating the lcd functioning code

4.1.3 Communication Module

The nRF radio module was chosen. It uses SPI(Serial peripheral interface). The code written was based off of dparsons NRFLite code with removal of several lines of code that were used for debugging. Programming the nRF module mostly consists of reading and writing from registers so as to execute commands. For example, code to send a packet involves, preparing the module for transmission, clearing flags, sending the packet via SPI, waiting for the packet to be sent, returning a result that represents whether the packet was successfully sent or not. The code snippet below demonstrates this: Due to challenges during receiving, it was required that a uniform packet size be chosen so as to be able to receive all types of packets. The packet structure of the end-to-end packets was thus employed as the default packet structure (as it is the largest). For the other packets, after the payload has been encoded in the prefix of the packet, the suffix is set to zeros.Below is

```
uint8_t send(uint8_t toRadioId, void *data, uint8_t length, SendType sendType){
    prepForTx(toRadioId, sendType);

    // Clear any previously asserted TX success or max retries flags.
    writeRegisterName(STATUS_NRF, (1 << TX_DS) | (1 << MAX_RT));

    // Add data to the TX buffer, with or without an ACK request.
    if (sendType == NO_ACK) { spiTransfer(WRITE_OPERATION, W_TX_PAYLOAD_NO_ACK,
data, length); }
    else                    { spiTransfer(WRITE_OPERATION, W_TX_PAYLOAD       ,
data, length); }

    uint8_t result = waitForTxToComplete();
    return result;
}
```

Figure 4.5: Code snippet of sending code

a code snippet showing the packet structure of the packets used. The process of encryption for

```
typedef struct{
    RadioPacketType packetType;
    uint8_t intended_final_destination;
    uint8_t from_radio_id;
    uint8_t msg[20];
}EndUserMsg;
```

Figure 4.6: Packet Structure for packets to be used

transmission was as follows:

1. Combine 4 8-bit bytes to form a 32-bit integer
2. Perform end-to-end encryption using Diffie-Hellman algorithm
3. Perform encryption to base station using RSA algorithm
4. Splitting the encrypted integer into 8-bit bytes and placing them in the packet
5. Filling the remainder of the packet with zeros.

Special care has to be taken, in particular when placing the bytes into the packet, it is important to not which part goes first. The code snippet below shows the process performed. It is important to note that when encrypting the packets are read in order however when encapsulating they are inserted in reverse order. While it was intended to use multiple channels for communication, unfortunately the code used on the server side could not perform the channel switching. Thus the channel was set to a default value of channel 100 (corresponds to a frequency of 2500MHz). This value was chosen as it had least interference at the area of testing. It is the channel used by all devices including the base station when sending the authentication details. Farther analysis was performed to ensure that the base station could handle the load on a single channel as the speed was set to a value of 1MBps. The initialization process involved setting the registers to the required channel and setting the output power to 0dBm. The other option that is available are output speed of 2MBps. It however requires more power to transmit at 0dBm with such a high speed. In case a particular channel has a lot of interference the channels can be scanned and a different channel chosen.

4.1.4 Modular Arithmetic

As encryption and decryption is done 4 bytes at a time, a minimum of 32-bits is required in order to successfully encrypt and decipher packets. A C-style struct was used to perform modular

```
243         while(i < msg_length){
244                 uint32_t to_encrypt = 0;
245                 //read 4 bytes at a time
246                 for(uint8_t j = 0; j < 4; j++,i++){
247                         to_encrypt <<= 8;
248                         to_encrypt |= typed_message[i];
249                 }
250                 //arr[0] = 0;
251                 //my_itoa(i,arr);
252                 //at this point i is 4!!!!!!!!
253                 //encrypt using diffie hellman first
254                 to_encrypt = encrypt_diffie(to_encrypt,num_to_send);
255                 //encrypt using rsa
256                 to_encrypt = encrypt_rsa(to_encrypt,mod_bs_rsa,encrypt_bs_rsa);
257                 //break down the encrypted message and repackage it into byte
    sized segments
258                 for(uint8_t j = 0; j < 4; ++j){
259                         msg_packet.msg[i-j-1] = (to_encrypt & 0xFFUL);
260                         to_encrypt >>= 8;
261                 }
262                 //arr[0] = 0;
263                 //my_itoa(i,arr);
264         }
265         for(;i < 20; ++i)msg_packet.msg[i] = 0;
```

Figure 4.7: Process of encapsulation

arithmetic. The definition of the struct is shown below: It stored two values, the integer value and

```
10 typedef struct {
11     uint32_t num;
12     uint64_t mod;
13 }modint;
```

Figure 4.8: Definition of modulo integer struct

the modulo-base used in the definition. The arithmetic operations of addition and subtraction were carried out normally with the addition that the result that was stored was the remainder upon division by mod. Since negative values are not included in finite fields, when the difference is negative, the value of mod is added so as to bring the result within the range of $[0, mod - 1]$. The implementation of the two operations (addition and subtraction) is shown below. Multiplication and division

```
42 modint add (modint lhs, modint rhs){
43     uint64_t mod = lhs.mod;
44     modint ans;
45     uint32_t sum = lhs.num + rhs.num;
46     while(sum > mod){
47         sum -= mod;
48     }
49     from_int(sum,&ans,mod);
50     return ans;
51 }
52
53 modint sub(modint lhs, modint rhs){ // lhs - rhs;
54     uint64_t mod = lhs.mod;
55     modint ans;
56     uint32_t diff;
57     if(lhs.num < rhs.num){
58         diff = mod + lhs.num - rhs.num;
59     }
60     else{
61         diff = lhs.num - rhs.num;
62     }
63     from_int(diff,&ans,mod);
64     return ans;
65 }
```

Figure 4.9: Implementation of addition and subtraction with modulo

were slightly more involving. When implementing multiplication, it is important to note that the product of two 32 bit integers could be 64 bits. Thus, even before modulo operation is applied, the

result could easily overflow a 32 bit storage. In order to combat this, a temporary storage of 64 bits should be used to store the immediate product. The remainder upon division by mod is then taken and is stored as the result. This can be done in a single line in C using type casting. The implementation is shown below: Before examining division, it is easier to examine exponentiation. Naive

```
67 modint mul(modint lhs, modint rhs){
68     uint64_t mod = lhs.mod;
69     modint ans;
70     uint32_t prod = ((uint64_t)lhs.num * rhs.num) % mod;
71     from_int(prod,&ans,mod);
72     return ans;
73 }
```

Figure 4.10: Implementation of multiplication with modulo

implementation of exponentiation results in time complexity of $O(n)$. This is too slow especially when dealing with large numbers as is the case in Diffie-Hellman and RSA. An faster algorithm known as binary exponentiation can be performed so as to improve the complexity to $O(log(n))$. The details of the particular algorithm were discussed in chapter 2. The code snippet below shows an implementation of binary exponentiation: Division is even more involved. It is implemented

```
75 modint binpow(modint b,uint64_t exp){
76     uint64_t mod = b.mod;
77     modint res;
78     from_int(1,&res,mod);
79     while(exp){
80         if(exp & 1){
81             res = mul(res,b);
82         }
83         b = mul(b,b);
84         exp >>= 1;
85     }
86     return res;
87 }
```

Figure 4.11: Binary exponentiation

using the concept of multiplicative inverses. The multiplicative inverse of a number a modulo m is defined as the number b such that the product $ab \equiv 1 \ mod \ m$. The multiplicative inverse does not always exist. There are two main methods of getting the multiplicative inverse. The first is using Euclid's extended GCD algorithm, and the second is using binary exponentiation. The second algorithm however only works for prime fields where the multiplicative inverse always exists. This can be proved using Fermat's little theorem. In the second technique, the number whose modulo inverse is required is simply raised to the value of the prime modulo - 2.That is $a^{-1} \equiv a^{m-2} \ mod \ m$ when m is prime. This can easily be proved using Fermat's little theorem. The first technique can be used for all numbers but the second one can only be used for prime numbers. The first technique was thus used for RSA and the second for Diffie-Hellman. An implementation of both techniques is shown below: Using the multiplicative inverse, division is thus defined. That is $\frac{a}{b} = a \cdot b^{-1}$

```
89 modint modinv(modint a){
90     uint64_t mod = a.mod;
91     modint res = binpow(a,mod - 2);
92     return res;
93 }
```

Figure 4.12: Implementation of modulo inverse with binary exponentiation

```
int extended_euclid(int a, int b, int * x, int * y){//get gcd using extended
euclid algorithm
    if (a == 0){
        (*x) = 0;(*y) = 1;
        return b;
    }
    int x_tmp,y_tmp;
    int g = extended_euclid(b % a, a, &x_tmp, &y_tmp);
    *x = y_tmp - (b / a) * x_tmp;
    *y = x_tmp;

    return g;
}
```

Figure 4.13: Modulo inverse with extended Euclid algorithm

where b^{-1} is the multiplicative inverse of b modulo m. The implementation is shown below:

```
modint divide(modint a, modint b){// a/b
    modint res = mul(a, modinv(b));
    return res;
}

```

Figure 4.14: Division with modulo

4.1.5 RSA

RSA was implemented using the modulo integer library. The formulas implemented are those that were discussed in chapter 2. The implementation is shown below: For encryption, the public keys

```
uint32_t encrypt_rsa(uint32_t data,uint32_t n, uint32_t e){
    modint a;
    from_int(data,&a,n);
    modint d = binpow(a,e);
    //return 328509;
    return d.num;
}
uint32_t decrypt_rsa(uint32_t data){
    modint a;
    from_int(data,&a,mod_rsa_me);
    modint m = binpow(a,decipher_key_rsa);
    return m.num;
}
```

Figure 4.15: implementation of RSA encryption and decryption

n and e are passed as arguments while for decryption the private key d is used. It is hard coded as a value. Since for RSA it is required that the message be smaller than public key n the value of n has to be at least 32 bits long (as encoding is done 4 bytes at a time). The two numbers, r and s that make up n must thus be around 16 bits long. The table below shows a list of prime numbers that may be used. Any two of the above values could be chosen. It is also sufficient to choose one prime around 2^{15} and another around 2^{17}. Once the two primes have been chosen, the value of n is computed. A value of e is then determined at random. Euclid's algorithm is then used to determine if there exists a modulo inverse of e modulo d. If it does not exist, another value is chosen for e. Once the value of d is found, it is set as the private key.

Expression	Value
$2^{16} - 15$	65521
$2^{16} - 17$	65519
$2^{16} - 39$	65497
$2^{16} - 57$	65479
$2^{16} - 87$	65449
$2^{16} - 89$	65447
$2^{16} - 99$	65437
$2^{16} - 113$	65423
$2^{16} - 117$	65419
$2^{16} - 123$	65413

Table 4.1: Prime numbers close to 2^{16}

4.1.6 Diffie-Hellman

Diffie-Hellman was also implemented using the modulo integer library. Once the keys had been exchanged, modulo multiplication was used to encrypt and division to decipher the messages. This encryption technique is however not the best. A more appropriate symmetric algorithm such as AES would be preferred. Immediately the system is turned on, the Diffie-Hellman keys, X_i and Y_i are initialized. The value of X_i is hard coded and from it the value of Y_i is determined. The code snippet below shows the implementation. The key Y_i is the one which is sent to initialize end-to-

```
24 void init_diffie(){
25     modint a;
26     from_int(generator,&a,mod);
27     modint b = binpow(a,X_i);
28     Y_i = b.num;
29 }
```

Figure 4.16: Diffie-Hellman initialization

end encryption between end equipment. The encryption key and decipher key are then computed. Once this is accomplished the process of encryption and decryption simply involve multiplication and decryption as has been discussed above. The implementation is shown below.

4.2 Hardware Implementation

For the end-equipment, a 9V supply was used to power the device. Since none of the device works with a supply of 9V, the supply was placed at the input terminals of the buck-converter. The

```
30 //when a device sends its Y_i initialize communication with them
31 void init_endequip_diffie(uint8_t num,uint32_t Y_theirs){
32     modint a;
33     from_int(Y_theirs,&a,mod);
34     modint b = binpow(a,X_i);
35     encryption_keys_diffie[num] = b.num;
36     a = modinv(b);
37     decryption_keys_diffie[num] = a.num;
38 }
```

Figure 4.17: End-to-End Diffie Hellman initialization

```
//encrypt packet. Uses modulo multiplication
uint32_t encrypt_diffie(uint32_t data,uint8_t num){
    modint a,b;
    from_int(data,&a,mod);
    from_int(encryption_keys_diffie[num],&b,mod);
    modint d = mul(a,b);
    return d.num;
}
//decrypt packet. Uses modulo multiplication by modulo inverse
uint32_t decrypt_diffie(uint32_t data, uint8_t num){
    modint a,b;
    from_int(data,&a,mod);
    from_int(decryption_keys_diffie[num],&b,mod);
    modint d = mul(a,b);
    return d.num;
}
```

Figure 4.18: Encryption and Decryption for end to end communication

buck converter potentiometer was then set to provide 5V across its output terminal. This 5V was connected to both the Atmega32, the LCD and the 3.3V linear regulator. The 3.3V supply was connected to the NRF radio module. On the Atmega32, the following pins were used: The IRQ

PIN	**FUNCTION**
4	Connected to chip enable of NRF module
5	Connected to chip select of NRF module
6	Connected to MOSI of NRF module
7	Connected to MISO of NRF module
8	Connected to SCK of NRF module
10	Connected to 5V supply
11	Connected to GND
(14-17)	Connected to columns (1 to 4) of keypad
(18-21)	Connected to rows (1 to 4) of keypad
22	Connected to SCL of PCF85474 of LCD display
23	Connected to SDA of PCF85474 LCD display

Table 4.2: Pin connection for ATMEGA32 MCU in end equipment

pin of the NRF module was not connected as interrupts were not used in this implementation.

For the base station, an Arduino Uno was used. The NRF radio module supply pins were connected to the 3.3V and the GND pins.The base station was then connected to a laptop. This provided two functions, powering both the NRF and the Arduino and facilitating communication between the laptop and the arduino. The operation of the base station was monitored through the serial monitor of the Arduino IDE. The other pins of the NRF module were connected as follows:

PIN ON NRF	PIN ON ARDUINO
CE	9
CSN	10
MISO	12
MOSI	11
IRQ	7
SCK	13

Table 4.3: Pin connection for NRF radio module in Base station

Chapter 5: Results and Analysis

The resulting system was analysed in relation to the following quality of service parameters:

1. Latency
2. Complexity
3. Data Rates
4. Range

5.1 Latency

Latency in general is defined as the time delay between a cause and its effect. In communication systems, this is the time between when the message was sent and when it was received and decoded. It includes the time taken for the message to be encoded, to propagate on the communication channel and to be decoded by the receiver. For perfect communication channels, the latency is only dependent on the propagation speed of the signal in the channel. However, for coded systems, it is farther dependant on the encoding and decoding time. For the developed system, the latency will be dependant on the time taken to perform encryption and decryption.

In order to measure the latency due to encryption and decryption, code was written to encrypt one thousand messages and the time it took was measured. Let the result be $T_{latency}$. The time taken to encrypt a single message was then obtained as $\frac{T_{latency}}{1000}$. The code snippet that was used is shown below: After running the above code, it was found that it took about 25 seconds in order to encrypt

```
int main(){
    init_main_timer();
    lcd_init();
    init_diffie();
    init_endequip_diffie(RADIO_ID,Y_i);
    lcd_print("begin: ");
    my_itoa(main_timer_now());
    lcd_print(string_rep);
    for(uint32_t i = 0; i < 1000; ++i){
        uint32_t _ = encrypt_diffie(i,RADIO_ID);
        _ = encrypt_rsa(_,mod_bs_rsa,encrypt_bs_rsa);
    }
    lcd_second_line();
    lcd_print("end: ");
    my_itoa(main_timer_now());
    lcd_print(string_rep);
    while(1){

    }
    return 0;
}
```

Figure 5.1: Code that was used to measure latency

the messages. Thus, in order to encrypt a single message, it would take about 0.025 seconds.

5.2 Complexity

Complexity can be analysed in terms of computational complexity and the hardware complexity. For computational complexity, big-Oh, analysis is used to compute the time and hardware complexity. For the input, the number of computational steps is dependant on the size of the input device since for each row, a pass is made through each column. This is a fixed value for the implementation and thus the complexity of the input is $O(1)$. However, as the keyboard size increases, this may change to $O(r.c)$ where r is the number of rows and c is the number of columns. A similar analysis can be made for the output devices with the addition that each pixel is also considered.

For the modular arithmetic operations, if we consider the inbuilt arithmetic operations to be constant time (this is not always the case since as the number of bits in the numbers increases, the time per operation also increases) then multiplication, addition and subtraction will be $O(1)$, exponentiation and division will be $O(loge)$ where e is the value of the exponent. If larger integer sizes are to be used, e.g. 512 bit integers, then the size of the integers becomes significant and the operation now become $O(n)$ for addition and subtraction, $O(n^2)$ for multiplication and $O(n^3)$ for exponentiation and division where n is the number of bits.

Thus, Diffie-Hellman key exchange which consists of one exponentiation for the key exchange and one division had a complexity of $O(loge)$ for both the initialisation and the encryption. Similarly, RSA encryption and decryption which both consisted of one exponentiation had a complexity of $O(loge)$ as well

5.3 Data Rates

The NRF radio module allows communication at 2MBps and at 1 MBps. For the implementation, the communication speed was selected to be 1Mbps. Since the complete packet may be up to 23 bytes and 3 bytes are required for communication, the total amount of user data that may be communicated is roughly $\frac{20}{23}$ the transmitted data. This gives data rates of about 850kBps. When the channel quality is low, a lot of retransmission is required which significantly lowers the data rates. For very poor channel the data rate may go as low as 100kBps (if 10 retransmissions are required. If more than 10 retransmissions are required, the communication automatically fails in order to save power.

5.4 Range

The range gives a measure of how far a node could be from the base station and still manage to communicate. The range is affected by the presence of obstacles such as walls in buildings. The table below gives the measurements that were taken. Another factor that affects the range is the power levels of the batteries. Since communication is to be done at 0dBm, if the batteries do not supply enough power to the end equipment devices, they will not transmit at the required 0dBm and thus the sent message will not be received. This can be alleviated by using chargeable batteries.

Distance	Obstacles	Successful Transmission
5 m	None	Yes
5 m	1 wall (thickness ∼30cm)	Yes
10 m	None	Yes
10 m	3 walls	Yes
15 m	None	Yes
15 m	2 walls	No
50 m	None	Yes
50 m	1 building	No
100 m	None	Yes
100m	Several buildings	No

Table 5.1: Results of range measurement

5.5 Demonstration of Operation

5.5.1 Joining Cell

As has been discussed, when a user joins the cell, they send an encrypted packet to the base station that contains the public keys of that device. The screenshot below shows the base station having received the public key details of the end equipment device that is joining.

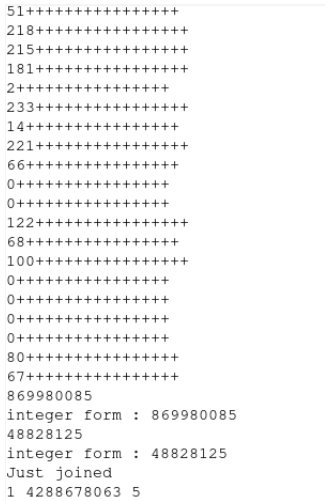

Figure 5.2: Screenshot of Base Station receiving joining instructions

The values with +'s proceeding are the individual bytes that were received. When these bytes were concatenated, they corresponded to the integers $869,980,085$ and $48,828,125$. These values correspond to the public keys of the particular end equipment. They have been encrypted with RSA encryption with the base station's public keys thus do not bare the required meaning. The base station thus decrypts the values to obtain $4,288,678,063$ and 5. These are the required public keys for the base station . The final line represents the received values i.e. the end equipment's ID and the public key for the end equipment.

5.5.2 Sending End to End Encryption Initialisation

In order to begin end to end encryption, the communicating end equipment each send their public key Y_i to each other. The base station can observe these keys but can't combine them to obtain their communicating keys. The diagram below shows the observed values at the terminal of the base station.

```
integer form : 1881549477
after decryption2114811280
after encryption1
what was received 16777216
16777216
integer form : 16777216
after decryption1580548234
after encryption1
what was received 0
0
```

Figure 5.3: Screenshot of Base Station receiving end-to-end encryption initialization

The above diagram shows the entire transmitted packet and contains the preamble instructions including the sender's ID, and the type of packet. Thus it is not simple to obtain the actual key value. However, even if one put in the effort to separate the preamble from the key, it would still not be as effective for them as they need the private key.

5.5.3 Sending Message

When sending a message, the user types the message and the ID of the receiving device. An example is shown below:

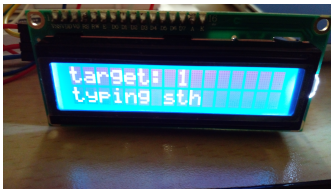

Figure 5.4: Typing the equipment ID of receiver

Figure 5.5: Typing message to send

In the above example, end equipment with ID 2 is to send a message to the end equipment with ID 1. The message is to read "hello world". The receiving end ID is denoted by target. The packet that is received in the base station is shown below:

```
what was received 1273345056
1273345056
integer form : 1273345056
after decryption609170201
after encryption212152891
what was received 3602738793
3602738793
integer form : 3602738793
after decryption233057650
after encryption1196526115
what was received 418188987
418188987
integer form : 418188987
after decryption769594734
after encryption2381589284
```

Figure 5.6: Screenshot of Base Station receiving end-to-end encrypted message

As can be seen from above, even after the base station decrypts the message using RSA, the resultant values are not easily decipherable. Since the packet also contains the preamble information, it is difficult to separate and then decipher what has been received. At the receiving end, (i.e end equipment 1) the correct message is received and deciphered.

Figure 5.7: Received message at receiving end

From the above, both the sending end ID and the sent message can be seen. It should be noted that this is the message that was transmitted thus successful encryption and secure transmission has been done.

Chapter 6: Conclusion and Recommendations

6.1 Conclusion

From the results and analysis, a working prototype of a communication system was built. It was observed that the user data could be encrypted and decrypted successfully. However, if an error occured during the transmission of the Diffie-Hellman keys, it could result in an inability of the end-equipment devices to communicate. This can be solved by restarting the devices. This can be solved by continuously confirming the keys but would result in increased latency.

Within buildings and offices, it is possible to use the system for communication by having one router to cover 50 meters in the hallway. This will come at a relatively low cost and will allow sending and receiving of messages. However for streaming of large quantity of data such as is needed during watching of videos, it will not be possible since the latency is high.

6.2 Recommendation

The system can be improved by allowing it to perform video and audio communication. This will enable features such as video conferencing and phone calls. Also in order to improve on the security, big integers may be implemented (500 - 2000 bit integers), while this will reduce the latency, it will make it even more difficult to break the encryption. Another addition that can be made is using a base station with more memory such as a Raspberry Pi so that it can resend messages that do not get delivered because the end equipment is off during the transmission.

Bibliography

[1] Wireless and Mobile Network Security Basics, Security in On-the-shelf and emerging Technologies by Hakima Chaouchi, Maryline Laurent-Maknavicius

[2] Threat modelling framework for mobile communication systems Siddarath Prakash Rao, Silke Holtmanns, Tuomas Aura

[3] A Survery of Cryptographic Methods in Mobile Network Technologies from 1G to 4G by Fredrick Njoroge , Lincoln Kamau.

[4] Encryption of 4G mobile broadband methods,Reihaneh Vafaei

[5] C. Hanser, S. Moritz, F. Zaloshnja and Q. Zhang, "Security in Mobile Telephony: The Security Levels in the Different Handy Generations," Uppsala Universitet, Uppsala, 2014.

[6] Authentication and Secure Communication in GSM, GPRS, and UMTS Using Asymmetric Cryptography Wilayat Khan, Habib Ullah.

[7] A5 encryption in GSM Oliver Damsgaard Jensen Kristoffer Alvern Andersen

[8] A comparative introduction to 4G and 5G technology cablelabs https://www.cablelabs.com/insights/a-comparative-introduction-to-4g-and-5g-authentication

[9] The Verizon Wireless 4G - LTE Network: Transforming business with next generation Technology. Verizon wireless

[10] Cryptography based Authentication Methods. Mohammed Alia,Abdelfatah Aref Tamimi, and Omaima N. A. AL-Allaf

[11] A Method for Obtaining Digital Signatures and Public-Key Cryptosystems R.L. Rivest, A. Shamir, and L. Adleman

[12] New Directions in Cryptography Whitfield Diffie and Martin E. Hellman IEEE transactions on Information Theory November 1976.

[13] github.com/AlvyneZ/Access-Control

[14] Guide to Bluetooth Security: Recommendation of the National Institute of Science and Technology Special Publication 800-121John Padgette Karen Scarfone Lily Chen.

[15] The International Conference on Advanced Wireless, Information, and Communication Technologies (AWICT 2015) Wi-Fi security analysis Tahar Mekhaznia, Abdelmadjid Zidani

[16] IOSR Journal of Computer Engineering (IOSR-JCE) An Overview of the RC4 Algorithm Isnar Sumartono et al.

[17] Symmetric and Asymmetric Encryption Gustavus J. Simmons

[18] A review on Symmetric Key Encryption Techniques in Cryptography Mohammad Ubaidullah Bokhari, Qahtan Makki Shallal

[19] A personal view on average case complexity Russel Impagliazzo

[20] 16th ESICUP Meeting, ITAM, Mexico City Knapsack Problem and variants Michele Monaci

[21] Modern Integer Factorization Techniques Gireesh Pandey and S K Pal

[22] Rosen, Kenneth H. (2011). Elementary Number Theory and Its Application (6th ed.). Pearson. p. 368.

Appendices

Appendix A: End Equipment Code

As the code is several of lines long, only some of the crucial parts have been outlined here. The rest of the code may be viewed at: https://github.com/mpily/secure_rf_comm

A.1 Modular Arithmetic Library

```
/*
    Author : mpily
    modulo integers to be used in encryption
*/

#ifndef MODINT
#define MODINT
#include<stdint.h>
#include<stdio.h>
typedef struct {
    uint32_t num;
    uint64_t mod;
}modint;
////////////////////////////////////////////////////////////
//function definitions
void from_int(uint32_t a, modint * b,uint64_t mod);//constuct from integer
void print(modint * a); // printing function
//arithmetic operations modulo mod
modint add(modint lhs, modint rhs);
modint sub(modint lhs, modint rhs);
modint mul(modint lhs, modint rhs);
modint binpow(modint b, uint64_t exp);// b ^ exp % mod
modint modinv(modint a);
modint divide(modint a, modint b);// a * b^-1 % mod
// bitwise operation
uint32_t xor_mask(modint a, uint32_t val);//a xor val
////////////////////////////////////////////////////////////

void from_int(uint32_t a, modint * b,uint64_t mod){
    b->mod = mod;
    if(a < mod){
        b-> num = a;
    }
    else{
        b->num = a % mod;
    }
}
```

```
void print(modint * a){
    printf("%d", a -> num);
}

modint add (modint lhs, modint rhs){
    uint64_t mod = lhs.mod;
    modint ans;
    uint32_t sum = lhs.num + rhs.num;
    while(sum > mod){
        sum -= mod;
    }
    from_int(sum,&ans,mod);
    return ans;
}

modint sub(modint lhs, modint rhs){ // lhs - rhs;
    uint64_t mod = lhs.mod;
    modint ans;
    uint32_t diff;
    if(lhs.num < rhs.num){
        diff = mod + lhs.num - rhs.num;
    }
    else{
        diff = lhs.num - rhs.num;
    }
    from_int(diff,&ans,mod);
    return ans;
}

modint mul(modint lhs, modint rhs){
    uint64_t mod = lhs.mod;
    modint ans;
    uint32_t prod = ((uint64_t)lhs.num * rhs.num) % mod;
    from_int(prod,&ans,mod);
    return ans;
}

modint binpow(modint b,uint64_t exp){
    uint64_t mod = b.mod;
    modint res;
    from_int(1,&res,mod);
    while(exp){
        if(exp & 1){
            res = mul(res,b);
        }
        b = mul(b,b);
        exp >>= 1;
```

```
    }
    return res;
}

modint modinv(modint a){
    uint64_t mod = a.mod;
    modint res = binpow(a,mod - 2);
    return res;
}
modint divide(modint a, modint b){// a/b
    modint res = mul(a, modinv(b));
    return res;
}

uint32_t xor_mask(modint a, uint32_t val){
    uint32_t masked = val ^ a.num;
    return masked;
}
#endif
```

A.2 RSA Encryption

```
/*
    used to implement rsa algorithm for encryption
    two primes being used
    example:
    r = 12809 s = 18587
    n = r.s = 238080883
    e = 3
    d = modinv 3 mod 238049488 = 158699659
    Encryption
    C = M ^ e
    Decryption = C ^ d
*/
#ifndef RSA
#define RSA
#include"../bigint/modint.h"
/*
encryption keys for base station
*/
const uint32_t mod_bs_rsa = 4292870399ULL;
const uint32_t encrypt_bs_rsa = 11;


const uint32_t mod_rsa_me = 4288678063ULL;
```

```
const uint32_t decipher_key_rsa = 2573128253ULL;
const uint32_t encrypt_key_rsa = 5;
uint32_t encrypt_rsa(uint32_t data,uint32_t n, uint32_t e){
    modint a;
    from_int(data,&a,n);
    modint d = binpow(a,e);
    //return 328509;
    return d.num;
}
uint32_t decrypt_rsa(uint32_t data){
    modint a;
    from_int(data,&a,mod_rsa_me);
    modint m = binpow(a,decipher_key_rsa);
    return m.num;
}
#endif
```

A.3 Diffie-Hellman Key Exchange

```
/*
    Used to implement Diffie Hellman Key Exchange
    Step 1. Generate X_i
    Step 2. Compute Y_i
    Step 3. Store X_i and Y_i
    Step 4. When other device sends Y_theirs, Compute key = Y_theirs ^ X_i
    Step 5. Compute modulo inverse of key.
    Step 6. Encryption is Modulo multiplication
    Step 7. Decryption is Multiplying by modulo inverses
*/
#ifndef DIFFIE
#define DIFFIE
#include"../bigint/modint.h"
//keys and mods and generator
const uint64_t mod = 2147483647ULL;
const uint32_t generator = 16807ULL;
uint32_t encryption_keys_diffie[128];
uint32_t decryption_keys_diffie[128];
/////personal Y_i and X_i
const uint32_t X_i = 31489ULL;
uint32_t Y_i;
//////////////////functions///////////////////////
// initializing personal keys;
void init_diffie(){
    modint a;
    from_int(generator,&a,mod);
```

```
    modint b = binpow(a,X_i);
    Y_i = b.num;
}
//when a device sends its Y_i initialize communication with them
void init_endequip_diffie(uint8_t num,uint32_t Y_theirs){
    modint a;
    from_int(Y_theirs,&a,mod);
    modint b = binpow(a,X_i);
    encryption_keys_diffie[num] = b.num;
    a = modinv(b);
    decryption_keys_diffie[num] = a.num;
}
//encrypt packet. Uses modulo multiplication
uint32_t encrypt_diffie(uint32_t data,uint8_t num){
    modint a,b;
    from_int(data,&a,mod);
    from_int(encryption_keys_diffie[num],&b,mod);
    modint d = mul(a,b);
    return d.num;
}
//decrypt packet. Uses modulo multiplication by modulo inverse
uint32_t decrypt_diffie(uint32_t data, uint8_t num){
    modint a,b;
    from_int(data,&a,mod);
    from_int(decryption_keys_diffie[num],&b,mod);
    modint d = mul(a,b);
    return d.num;
}
#endif
```

A.4 Overall Operating Code

```
#ifndef F_CPU
    #define F_CPU 8000000UL
#endif
#include<avr/io.h>
#include"nrf/NRFLite.h"
#include"lcd/lcd_i2c.h"
#include"lcd/main_timer.h"
#include"rsa/rsa.h"
#include"keypad/keypad.h"
#include"diffie-hellman/diffie.h"
#include<stdlib.h>
const uint8_t RADIO_ID = 2;
const uint8_t PIN_RADIO_CE = 3;
```

```
const uint8_t PIN_RADIO_CSN = 4;
uint8_t DESTINATION_RADIO_ID = 0;

//////////////////////////
//uint8_t radio_init(uint8_t radioId, uint8_t cePin, uint8_t csnPin, Bitrates
    bitrate, uint8_t channel, uint8_t callSpiBegin);
typedef enum{INITIALIZE, CHANNEL_CONTROL, E2EENCRYPTINIT, E2EMSG}RadioPacketType;
typedef struct{
    RadioPacketType packetType;
   uint8_t intended_final_destination;
   uint8_t from_radio_id;
   uint32_t n;//private key
   uint32_t e;//private key
}InitPacket;//structure of initialization packet
typedef struct{
    RadioPacketType packetType;
    uint8_t intended_final_destination;
    uint8_t from_radio_id;
    uint8_t msg[20];
}ReceivedPacket;
typedef struct{
    RadioPacketType packetType;
    uint8_t intended_final_destination;
    uint8_t from_radio_id;
    uint8_t msg[20];
}EndUserMsg;
ReceivedPacket new_packet;//a newly received packet
char received_msg[40];//to store received messages
uint8_t received_id;//store the id of received message
uint8_t recently_received;
uint8_t await_channel_info = 0;
uint8_t await_diffie_info = 0;
////////////////////debugging functions //////////////////////////
/*void my_itoa(uint32_t num, char * arr){
    for(int i = 0; i < 10; ++i){
        arr[i] = 0;
    }
    int pos = 0;
    if(num == 0){
        arr[pos] = '0';
    }
    while(num){
        arr[pos] = '0' + (num % 10);
        num/= 10;
        pos ++;
    }
    pos --;
```

```
    int i = 0;
    while(i < pos){
        char tmp = arr[i];
        arr[i] = arr[pos];
        arr[pos] = tmp;
        pos --;
        i ++;
    }
    lcd_clear();
    lcd_print("--");
    lcd_print(arr);
    _delay_ms(500);
}*/
////////////////////////////////////////////////////////////
uint8_t send_init_message(){
    //lcd_clear();
    //lcd_print("sending init message");
    EndUserMsg initmessage;
    initmessage.packetType = INITIALIZE;
    initmessage.intended_final_destination = 0;
    initmessage.from_radio_id = RADIO_ID;
    uint32_t n = encrypt_rsa(mod_rsa_me,mod_bs_rsa,encrypt_bs_rsa);
    uint32_t e = encrypt_rsa(encrypt_key_rsa,mod_bs_rsa,encrypt_bs_rsa);
    int8_t i = 0;
    for(;i < 4; ++i){
      initmessage.msg[4 - i - 1] = (n & 0xFF);
      n >>= 8;
    }
    i = 0;
    for(; i < 4; ++i){
      initmessage.msg[8 - i - 1] = (e & 0xFF);
      e >>= 8;
    }
    if (send(DESTINATION_RADIO_ID, &initmessage,
        sizeof(initmessage),REQUIRE_ACK)) // 'send' puts the radio into Tx mode.
    {
        lcd_clear();
        lcd_print("...Success");
        return 1;//successfully initialized
    }
    else
    {
        lcd_clear();
        lcd_print("...Failed");
        return 0;//failed to initialize;

    }
```

```
    await_channel_info = 1;
    return 0;
}
void handle_channel_control(){//when a channel control packet is received
   uint32_t channel_info = 0;//will be 4 bytes long and is encrypted
    for(int i = 0; i < 4; ++i){
      channel_info <<= 8;
      channel_info |= new_packet.msg[i];
    }
    uint8_t comm_channel = decrypt_rsa(channel_info);
    char num[10];
    //my_itoa(comm_channel,num);
    switch_channels(comm_channel);
}
void handle_e2e_init(){//when end to end encryption initialization packet is
    received
   /*
      Packet structure :
         -4 bytes and they encode Y_theirs
         -1 byte whether they want a response
   */
   uint8_t their_id = new_packet.from_radio_id;
   //lcd_print("initing: ");
   char dbg_id[2] = {'0' + their_id,0};
   lcd_print(dbg_id);
   uint32_t Y_theirs = 0;
   for(int i = 0; i < 4; ++i){
      Y_theirs <<= 8;
      Y_theirs |= new_packet.msg[i];
   }
   Y_theirs = decrypt_rsa(Y_theirs);
   init_endequip_diffie(their_id,Y_theirs);
   if(new_packet.msg[4]){
      EndUserMsg init_msg;
      init_msg.packetType = E2EENCRYPTINIT;
      init_msg.intended_final_destination = their_id;
      init_msg.from_radio_id = RADIO_ID;
      uint32_t Y_send = encrypt_rsa(Y_i,mod_bs_rsa,encrypt_bs_rsa);
      for(int i = 3; i >= 0; --i){
         init_msg.msg[i] = (Y_send & 0xFF);
         Y_send >>= 8;
      }
      init_msg.msg[4] = 0;
      for(int i = 5; i < 20; ++i)init_msg.msg[i] = 0;
      send(DESTINATION_RADIO_ID, &init_msg, sizeof(init_msg),REQUIRE_ACK);
   }
   //lcd_clear();
```

```
   //lcd_print("done initing");
   //_delay_ms(500);
}
void handle_e2e_msg(){
   char arr[15];
   for(int i = 0; i < 15; ++i)arr[i] = 0;
   received_id = new_packet.from_radio_id;
   uint8_t i = 0;
   while(i < 20){
      uint32_t bytes = 0;
      for(int j = 0; j < 4; ++j,i++){
         bytes <<= 8;
         bytes |= new_packet.msg[i];
      }
      bytes = decrypt_rsa(bytes);
      bytes = decrypt_diffie(bytes,received_id);
      for(int j = 0; j < 4; ++j){
         uint8_t nxt_char = (bytes & 0xFFUL);
         received_msg[i - j - 1] = nxt_char;
         bytes >>= 8;
      }
   }
   received_msg[20] = 0;
   //recently_received = 1;
   lcd_clear();
   lcd_print("from : ");
   char tgt[3] = {'0' + received_id,0,0};
   lcd_print(tgt);
   lcd_second_line();
   lcd_print(received_msg);
   _delay_ms(1000);
}
void checkRadio(){
    while(hasData(0)){
        readData(&new_packet);
        if(new_packet.packetType == CHANNEL_CONTROL){
           handle_channel_control();
        }
        else if(new_packet.packetType == E2EENCRYPTINIT){
         handle_e2e_init();
        }
        else if(new_packet.packetType == E2EMSG){
         handle_e2e_msg();//has a lot of control stuff best to handle alone;
        }
    }
}
uint8_t init_e2e(uint8_t num){
```

```
   //lcd_clear();
   //lcd_print("sending e2e again");
   EndUserMsg init_msg;
   init_msg.packetType = E2EENCRYPTINIT;
   init_msg.intended_final_destination = num;
   init_msg.from_radio_id = RADIO_ID;
   uint32_t Y_send = encrypt_rsa(Y_i,mod_bs_rsa,encrypt_bs_rsa);
   //break down the encrypted message and repackage it into byte sized segments

   for(int i = 3; i >= 0; --i){
      init_msg.msg[i] = (Y_send & 0xFFUL);
      Y_send >>= 8;
   }
   if(encryption_keys_diffie[num] == 0)init_msg.msg[4] = 1;
   else init_msg.msg[4] = 0;
   for(int i = 5; i < 20; ++i)init_msg.msg[i] = 0;
   send(DESTINATION_RADIO_ID, &init_msg, sizeof(init_msg),REQUIRE_ACK);
   uint8_t attempts = 0;
   while(encryption_keys_diffie[num] == 0 && attempts < 10){
      checkRadio();
      //lcd_clear();
      //lcd_print("not yet");
      if(attempts % 5 == 0)
         send(DESTINATION_RADIO_ID, &init_msg, sizeof(init_msg),REQUIRE_ACK);
      checkRadio();
      _delay_ms(1000);//wait some time then try again
      checkRadio();
      attempts ++;
   }
   if(encryption_keys_diffie[num] == 0)return 0;
   else return 1;
   //lcd_clear();
   //lcd_print("success --> ");
   //char numstr[2] = {'0' + num, 0};
   //lcd_print(numstr);
   //_delay_ms(500);
}

void send_e2e_msg(){
   EndUserMsg msg_packet;
   msg_packet.packetType = E2EMSG;
   msg_packet.intended_final_destination = num_to_send;
   msg_packet.from_radio_id = RADIO_ID;
   uint8_t inited = 0;
   inited = init_e2e(num_to_send);
   if(!inited){
      lcd_clear();
```

```
    lcd_print("couldn't send");
    return;
  }
  msg_length += (4 - (msg_length % 4));//make message length divisible by 4
  //lcd_clear();
  //lcd_print("done init encrypt");
  //_delay_ms(1000);
  //char arr[20];
  //for(int i = 0; i < 20; ++i) arr[i] = 0;
  uint8_t i = 0;
  while(i < msg_length){
    uint32_t to_encrypt = 0;
    //read 4 bytes at a time
    for(uint8_t j = 0; j < 4; j++,i++){
      to_encrypt <<= 8;
      to_encrypt |= typed_message[i];
    }
    //arr[0] = 0;
    //my_itoa(i,arr);
    //at this point i is 4!!!!!!!!
    //encrypt using diffie hellman first
    to_encrypt = encrypt_diffie(to_encrypt,num_to_send);
    //encrypt using rsa
    to_encrypt = encrypt_rsa(to_encrypt,mod_bs_rsa,encrypt_bs_rsa);
    //break down the encrypted message and repackage it into byte sized segments
    for(uint8_t j = 0; j < 4; ++j){
      msg_packet.msg[i-j-1] = (to_encrypt & 0xFFUL);
      to_encrypt >>= 8;
    }
    //arr[0] = 0;
    //my_itoa(i,arr);
  }
  for(;i < 20; ++i)msg_packet.msg[i] = 0;
  msg_length = 0;
  checkRadio();
  uint8_t gone =
      send(DESTINATION_RADIO_ID,&msg_packet,sizeof(msg_packet),REQUIRE_ACK);
  uint8_t attempts = 0;
  while(!gone && attempts < 10){
    lcd_clear();
    lcd_print("message not yet sent");
    //checkRadio();
    gone =
        send(DESTINATION_RADIO_ID,&msg_packet,sizeof(msg_packet),REQUIRE_ACK);
    _delay_ms(300);
    attempts ++;
    checkRadio();
```

```
   }
   lcd_clear();
   lcd_print("message sent");
   checkRadio();//place in receive mode;
   _delay_ms(30);
}
void display_on_lcd(){
   if(updated && pressed_count){
      lcd_clear();
      if(typing_number){
         lcd_print("target: ");
         char tgt[3] = {'0' + num_to_send,0,0};
         lcd_print(tgt);
      }
      else{
         lcd_print(pressed_buttons);
      }
      updated = 0;
      lcd_second_line();
      lcd_print("typing sth");
   }
}

int main(){
    init_main_timer();
    lcd_init();
    init_keypad();
    init_diffie();
    init_endequip_diffie(RADIO_ID,Y_i);
    while(!radio_init(RADIO_ID, PIN_RADIO_CE,PIN_RADIO_CSN,BITRATE1MBPS,100,1)){
      lcd_clear();
        lcd_print("can't comm");
        //printDetails();
        //while(1);//wait here forever
    }
    while(!send_init_message()){
        _delay_ms(10);//wait for some time then try again.
    }
    while(1){
      //if(!await_channel_info && !await_diffie_info){
      check_keypad();
      //}
      if(msg_length){
         send_e2e_msg();
      }
      checkRadio();
      display_on_lcd();
```

```
    }
    return 0;
}
```

Appendix B: Base Station Code

This was implemented in one file on the Arduino IDE

```
/*

Demonstrates two-way communication without using acknowledgement data packets.
    This is much slower than
the hardware-based, acknowledgement data packet approach shown in the
    'TwoWayCom_HardwareBased' example,
but it is more flexible.

It is important to keep in mind the radio cannot send and receive data at the
    same time, instead, the
radio will either be in receiver mode or transmitter mode. NRFLite switches the
    mode of the radio in
hopefully a natural way based on the methods being called, e.g. 'send' puts the
    radio int32_to transmitter
mode while 'hasData' puts it int32_to receiver mode. But you may need to be
    aware of this mode switching
behavior when doing two-way communication. For example, you may need to minimize
    the amount of time the
radio is in transmitter mode if you would like it to participate in two-way
    communication with another
radio. An approach to minimize the amount of time the radio is a transmitter is
    shown in this example.

Software-based two-way communication is slow compared to the two-way
    communication support implemented
by the designers of the radio module in hardware. So check out the
    hardware-based two-way communication
example if you would like a faster solution.

Radio    Arduino
CE    -> 9
CSN   -> 10 (Hardware SPI SS)
MOSI  -> 11 (Hardware SPI MOSI)
MISO  -> 12 (Hardware SPI MISO)
SCK   -> 13 (Hardware SPI SCK)
IRQ   -> No connection
VCC   -> No more than 3.6 volts
GND   -> GND
```

```
*/

#include "SPI.h"
#include "NRFLite.h"
struct modint{
    uint32_t mod;
    uint32_t _num;
    modint(): _num(0){}
    modint(uint32_t num,uint32_t _mod){
        mod = _mod;
        num %= mod;
        while(num < 0)num += mod;
        _num = num;
    }
    modint & operator ++() {
        _num++;
        if(_num == mod) _num = 0;
        return *this;
    }
    modint & operator --(){
        _num --;
        if(_num < 0)
            _num += mod;
        return *this;
    }
    modint operator-- (int){
        modint ans = *this;
        --*this;
        return ans;
    }
    modint operator++ (int){
        modint ans = *this;
        ++*this;
        return ans;
    }

    modint& operator += (const modint & rhs){
        _num += rhs._num;
        while(_num >= mod) _num -= mod;
        return *this;
    }
    modint& operator -= (const modint & rhs){
        _num -= rhs._num;
        while (_num < 0) _num += mod;
        return *this;
    }
```

```
modint& operator *= (const modint& rhs){
    uint64_t ans = _num;
    ans *= rhs._num;
    _num = uint32_t(ans % mod);
    return *this;
}
modint& operator /= (const modint& rhs){return *this = *this * rhs.modinv();}
modint operator+() const {return *this;}
modint operator-() const {return modint() - *this; }
modint pow(uint64_t e) const {
    //assert(0 <= e);
    modint base = *this;
    modint ans(1,mod);
    while(e){
        if(e & 1){
            ans *= base;
        }
        base *= base;
        e >>= 1;
    }
    return ans;
}
modint modinv(){
    return pow(mod - 2);
}
friend modint operator +(const modint& lhs, const modint& rhs){
    modint ans = lhs;
    ans += rhs;
    return ans;
}
friend modint operator -(const modint& lhs, const modint& rhs){
    modint ans = lhs;
    ans -= rhs;
    return ans;
}
friend modint operator *(const modint& lhs, const modint& rhs){
    modint ans = lhs;
    ans *= rhs;
    return ans;
}
friend modint operator /(const modint& lhs, const modint& rhs){
    modint ans = lhs;
    ans /= rhs;
    return ans;
}
friend bool operator==(const modint& lhs, const modint& rhs){
    return lhs._num == rhs._num;
```

```
    }
    friend bool operator !=(const modint& lhs, const modint& rhs){
        return lhs._num != rhs._num;
    }
};
typedef modint mint;

const static uint8_t RADIO_ID = 0;
const static uint8_t DESTINATION_RADIO_ID = 0;
const static uint8_t PIN_RADIO_CE = 9;
const static uint8_t PIN_RADIO_CSN = 10;
typedef enum{INITIALIZE, CHANNEL_CONTROL, E2EENCRYPTINIT, E2EMSG}RadioPacketType;
typedef struct{
    RadioPacketType packetType;
    uint8_t intended_final_destination;
    uint8_t from_radio_id;
    uint32_t n;
    uint32_t e;
}fortests;
typedef struct{
    RadioPacketType packetType;
    uint8_t intended_final_destination;
    uint8_t from_radio_id;
    uint8_t msg[20];
}ReceivedPacket;
typedef struct{
    RadioPacketType packetType;
    uint8_t intended_final_destination;
    uint8_t from_radio_id;
    uint32_t channel;
}ChannelInfo;

typedef struct{
    uint32_t n,e,channel;
}endequipinfo;
endequipinfo details[100];
NRFLite _radio(Serial);
uint32_t _lastHeartbeatSendTime;
uint32_t _lastHeartbeatReceiveTime;

void setup()
{
    Serial.begin(115200);

    if (!_radio.init(RADIO_ID, PIN_RADIO_CE, PIN_RADIO_CSN,NRFLite::BITRATE1MBPS))
    {
        Serial.println("Cannot communicate with radio");
```

```
        while (1); // Wait here forever.
    }
}

void loop()
{
    // Send a heartbeat once every 4 seconds.
    //if (millis() - _lastHeartbeatSendTime > 3999)
    //{
       // _lastHeartbeatSendTime = millis();
        //sendHeartbeat();
    //}
    // Show any received data.
    checkRadio();
}
/*
    encryption keys unique to device
    //must satisfy
    ////////////////////set 1 - base station ////////////////////////
    r = 65521
    s = 65519
    n = r.s = 4292870399
    d = 11
    e = 1560996131

    ////////////////set 2 - UE1 ////////////////////////
    r = 65497
    s = 65479
    n = r.s = 4288678063
    d = 5
    e = 2573128253

    //////////////set 3 - UE2 ////////////////////////////
    r =
*/
uint32_t decrypt(uint32_t data)
{
    Serial.println(data);
    uint32_t x = data;
    Serial.print("integer form : ");
    Serial.println(x);
    mint a(x,4292870399ULL);
    mint ans = a.pow(1560996131ULL);
    return uint32_t(ans._num);
}
uint32_t encrypt(uint32_t n,uint32_t e,uint32_t value)
{
```

```
  uint32_t x = value;
  mint a(x,n);
  mint ans = a.pow(e);
  return uint32_t(ans._num);
}
void to_uint32(uint32_t & store, uint8_t * source)
{
  store = 0;
  for(int idx = 0; idx < 4; idx++)
  {
    store <<= 8;
    store |= source[idx];
  }
}
void to_uint8(uint32_t source, uint8_t * store)
{
  for(int idx = 0; idx < 4; idx++)
  {
    store[3 - idx] = source & 0xFF;
    source >>= 8;
  }
}
void handle_channel_control(ReceivedPacket radioData)
{
    uint8_t from = radioData.from_radio_id;
    for(int idx = 0; idx < 20; ++idx){
      Serial.print(radioData.msg[idx]);
      Serial.println("++++++++++++++++");
    }
    to_uint32(details[from].n,radioData.msg);
    to_uint32(details[from].e,radioData.msg + 4);
    details[from].n = decrypt(details[from].n);
    details[from].e = decrypt(details[from].e);
    ChannelInfo nwequip;
    nwequip.packetType = CHANNEL_CONTROL;
    nwequip.intended_final_destination = from;
    nwequip.from_radio_id = RADIO_ID;
    nwequip.channel = 100;
    nwequip.channel = encrypt(details[from].n,details[from].e,nwequip.channel);
    _radio.send(from, &nwequip, sizeof(nwequip));
    Serial.println("Just joined");
    Serial.print(from);
    Serial.print(" ");
    Serial.print(details[from].n);
    Serial.print(" ");
    Serial.print(details[from].e);
    Serial.println(" ");
```

```
}
void handle_end_to_end(ReceivedPacket radioData)
{
  Serial.println("should send something");
  ReceivedPacket final_packet;
  final_packet.packetType = radioData.packetType;
  final_packet.from_radio_id = radioData.from_radio_id;
  final_packet.intended_final_destination = radioData.intended_final_destination;
  if(details[radioData.intended_final_destination].n == 0)
  {
    return;
  }
  for(int idx = 0; idx < 20; idx += 4)
  {
    uint32_t intermediate;
    to_uint32(intermediate,radioData.msg + idx);
    Serial.print("what was received ");
    Serial.println(intermediate);
    intermediate = decrypt(intermediate);
    Serial.print("after decryption");
    Serial.println(intermediate);
    intermediate =
        encrypt(details[radioData.intended_final_destination].n,details[radioData.intended_final
    to_uint8(intermediate,final_packet.msg + idx);
    Serial.print("after encryption");
    Serial.println(intermediate);
  }
  for(int idx = 0; idx < 20; ++idx){
    Serial.println(final_packet.msg[idx]);
  }
  uint8_t count = 0;
  while(!_radio.send(radioData.intended_final_destination, &final_packet,
      sizeof(final_packet)) && count < 10){
    delay(100);
    count ++;
  }
  Serial.print("took ");
  Serial.print(count);
  Serial.print(" attempts");
}
void checkRadio()
{
    //Serial.println("Receiving Heartbeat");
    while (_radio.hasData()) // 'hasData' puts the radio into Rx mode.
    {
        ReceivedPacket radioData;
        _radio.readData(&radioData);
```

```
        //Serial.println(radioData.n);
        //Serial.println(radioData.e);
        if(radioData.packetType == INITIALIZE){
           handle_channel_control(radioData);
        }
        else{
           handle_end_to_end(radioData);
        }
    }
    //Serial.println("that's all");
}
```