LB/DON/106/2016
IT 01/133

Design and Development of an Efficient and Secure Lightweight Protocol for Wireless Sensor Networks

LIBRARY
UNIVERSITY OF MORATUWA, SRI LANKA
MORATUWA

K.Kesavan
(139168 U)

Dissertation submitted to the Faculty of Information Technology, University of Moratuwa, Sri Lanka, for the partial fulfilment of the requirements of the Degree of Master of Science in Information Technology.

004"16"
004 (043)

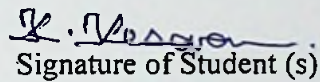
March 2016

TH 3168
+
1 DVD ROM
(TH 3160 - TH 3180)
TH 3168

Declaration

We declare that this thesis is our own work and has not been submitted in any form for another degree or diploma at any university or other institution of tertiary education. Information derived from the published or unpublished work of others has been acknowledged in the text and a list of references is given.

K. Kesavan.
Name of Student (s)

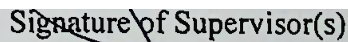

Signature of Student (s)

Date : 28/04/2016

Supervised by

M.F.M. Firdhous.
Name of Supervisor(s)

UOM Verified Signature


Signature of Supervisor(s)

Date : 28/04/2016

Dedicated to

my teacher Mr. S. Thirunavukkarasu

Acknowledgement

It should be mentioned with gratitude that many individuals influenced to accomplish this project in many ways.

First of all, I would like to respectfully express my sincere gratitude to my supervisor, Mr. M.F.M.Firdhous (Senior Lecturer), for his guidance, insightful suggestions, invaluable comments, and constant encouragement and optimism that not only helped me overcome frustrations and difficulties throughout my project, but also will influence my future career in pursuit of excellence. I was really fortunate to complete my project under his supervision.

I am also indebted to the evaluators of the interim report of this project, Dr. L. Ranathunga, Dr. C.R.J. Amalraj and Mr. B.H. Sudantha for their valuable advice and comments to improve my work.

As well, I am grateful to all of the academic staff of the Faculty of Information Technology who took lectures in our postgraduate programme, and specially the course coordinator – Mr.S.C.Premaratne, without their teaching and guidance, I could not have got the capacity to successfully complete this project. Further, the course – Literature Review & Thesis Writing taught by Prof. ASK Karunanda - has particularly supported very much to document this research work successfully.

Furthermore, I would like to extend my sincere thanks to my sectional head at the office – Eng. D.S.D.Jayasiriwardena (Additional General Manager) for his encouragement and support to pursue the postgraduate programme successfully. Additionally, I would like to thankfully acknowledge the financial support of the National Water Supply and Drainage Board to cover the fees of my first year programme.

Special heartiest thanks should go to my family, Ms. Bhavani, Mas.Ashvin and Mas. Avaneesh, who always showed me their unconditional love, faith and support during this task. Moreover, I would like to thanks to my friends who encouraged to perform the project.

Moreover, I would like to express my gratitude to Park & Miller, Carta, Marsaglia, University of Sheffield, GlibC, ANSI C and National Security Agency for utilizing their proposed pseudo random number generators and secure hash algorithms in this study.

Finally, I humbly bow before the almighty God for showering his blessings upon me and giving me the strength, wisdom and luck to reach this important milestone in my academic life.

Abstract

Water supply and sanitation services can no longer tolerate inefficiencies of their traditional non-intelligent distribution infrastructures due to the growth of demand for their uninterrupted services in quantity and quality wise. Wireless Sensor Networks can be employed to address these issues in a very cost effective manner. Already, Wireless Sensor Networks have been started to utilize in some countries for implementing their infrastructures of water supply and sanitation services as intelligent to provide better services and reduce financial losses. Ensuring efficient and secure data communication in Wireless Sensor Network is one of the major aspects in its wide range of applications. But, security solutions developed for traditional networks are not suitable for Wireless Sensor Networks due to its specific features. Many researches have been carried out to propose suitable efficient and secure lightweight protocols for Wireless Sensor Networks to improve their data communication.

In this project, an efficient and secure lightweight protocol has been proposed for Wireless Sensor Networks. Many literatures related to various security threats, security protocols and key management schemes of Wireless Sensor Networks have been critically reviewed at the beginning of the study. Literatures regarding the pseudo random number generators and hash algorithms relevant to these security architectures have also been critiqued to analyse the suitability of them.

In 2004, Park and Shin have proposed a lightweight protocol called Lightweight Security Protocol(LiSP). The salient feature of this protocol is the novel rekeying mechanism to tradeoff between security and resource consumption for large scale sensor networks. In 2006, Sun and coworkers have presented a lightweight security protocol with similar key management scheme of Park and Shin, but improved security mechanism by employing a pseudo random number generator - Linear Congruential Generator(LCG). In 2015, Jain and Ojha have identified that Park-Miller pseudo random number generator is better than Linear Congruential Generator for the lightweight security protocol. Further, in 2015, Ojha and Jain analysed some other pseudo random generators to evaluate the performances of the lightweight security protocol and concluded that Park-Miller pseudo random number generator is the most suitable one. But, these studies didn't consider Park-Miller's latest recommendation, or

other variations of the pseudo random number generators. Pseudo random number generators play vital role in the security and efficiency of the lightweight protocols. Moreover, It has been identified that Secure Hash Algorithm-1(SHA-1) employed in this protocol has similar effect as pseudo random number generator in the security and efficiency of the protocol.

Therefore, the performance of the lightweight protocol can be enhanced without compromising its security features, by utilizing more appropriate pseudo random number generator and hash function in its architecture.

So, the latter part of the project, the secure lightweight protocols having different pseudo random number generators and secure hash algorithms have been designed and implemented to evaluate their suitability for proposing an efficient and secure lightweight protocol.

Implementations have been modeled and evaluated in MATLAB software which had been recommended and utilized in many previous literatures for this purpose. Times taken for the computations have been analysed with pseudo random number generators and secure hash algorithms employed with their specific features.

The pseudo random number generator, LCG Sheffield, has been identified as a most suitable pseudo random number generator for the lightweight protocol. Secure Hash Algorithm -1 proposed in the previous studies has been identified as a most efficient hash function for the lightweight protocol.

This study proposes a secure lightweight protocol which is experimentally shown as, in average, 5.7% more efficient than the secure protocol proposed in the study by Jain and Ojha in 2015.

Table of Contents

Page No.

Abstract..v

List of Figures..xi

List of Tables..xiii

Abbreviations..xv

Chapter 1 – Introduction 1

1.1 Prolegomena 1

1.2 Background and Motivation..1

1.3 Problem definition..4

1.4 Aim and Objectives 4

1.4.1 Objectives related to the problem 5

1.4.2 Objectives related to the solution 5

1.5 Hypothesis 6

1.6 Resources required 6

1.7 Structure of the thesis 6

1.8 Summary...6

Chapter 2 – Developments in security of Wireless Sensor Network 7

2.1 Introduction 7

2.2 Review of Literature..7

2.3 Summarization of the reviews and Problem definition 12

2.3.1 Problem Definition 15

2.3.2 Identified Technologies 15

2.4 Summary...15

Chapter 3 – Technologies adopted for the proposed protocol..................16

3.1 Introduction..16

Page No.

3.2 Onetime pad(OTP) 16

3.3 Pseudo Random Number Generator (PRNG) 17

3.3.1 Linear Congruential Generator (LCG) 17

3.3.2 Park-Miller pseudo random number generator 18

3.3.3 Park-Miller-Carta pseudo random number generator 19

3.4 Block cipher 20

3.5 Dynamic key 20

3.6 Hash function 21

3.6.1 Secure Hash Algorithm -1 21

3.7 Symmetric key cryptosystems 22

3.8 Encryption primitives of cryptosystems 22

3.8.1 Addition operation 23

3.8.2 Subtract operation 23

3.8.3 XOR operation 23

3.8.4 Transpose operation 24

3.8.5 Swap operation 24

3.9 MATLAB 24

3.10 Summary 25

Chapter 4 – Approach for Efficient and Secure Lightweight Protocol 26

4.1 Introduction 26

4.2 Hypothesis 26

4.3 Input.......... 26

4.4 Output.......... 27

4.5 Process.......... 27

4.5.1 Encryption process..........27

4.5.2 Decryption Process 28

4.6 Features..........30

4.7 Users..........31

Page No.

4.8 Summary 31

Chapter 5 – Design of the Efficient and Secure Lightweight Protocol..................32

5.1 Introduction 32

5.2 Top Level Architecture of the Secure Lightweight Protocol 32

5.2.1 Generation of Dynamic Binary Key 32

5.2.2 Encryption Process 34

5.2.3 Decryption Process 35

5.3 Theoretical Analysis of the Architecture 39

5.3.1 Selection of PRNGs 39

5.3.2 Selection of Hash Functions 41

5.4 Summary 42

Chapter 6 – Implementation of the Efficient and Secure Lightweight Protocol 43

6.1 Introduction..43

6.2 Overview of the implementation..43

6.3 Software and Platform used for the implementation..........................43

6.4 Implementation of the module 'Generate Dynamic Binary Key' 44

6.5 Implementation of the function 'EnccryptionProcess' 46

6.6 Implementation of the function 'DecryptionProcess'.........................47

6.7 Implementation of the Module 'AutomateDataCollection_PRNG' 48

6.8 Implementation of the Module 'AutomateDataCollection_HF' 48

6.9 Implementation of the Module 'AutomateFunctionalityTesting' 49

6.10 Summary 49

Chapter 7 – Evaluation...50

7.1 Introduction 50

7.2 Testing the functionality of the Implementation 50

7.3 Evaluation Strategy 50

Page No.

7.4 Average execution times of the Implementations with different PRNGs 51

7.5 Average execution times of the implementations with different Hash Algorithms..........56

7.6 Summary 60

Chapter 8 – Conclusion and Further Work 61

8.1 Introduction 61

8.2 Conclusion 61

8.3 Further work 62

8.4 Summary 62

References 63

Appendix A – Detailed design Diagram 67

Appendix B – Selected Source Code 71

B.1 Listing 'AutomateDataCollection_PRNGEvaluation' (Only the specific modules) 71

B.2 Code Listing 'AutomateDataCollection_HF' (Only the specific modules) 93

Appendix C – Measured Execution Times..........104

Appendix D – Results of the Functionality Testing 113

D.1 Evaluating architectures having different PRNGs..........113

D.2 Evaluating architectures having different Hash Algorithms 115

D.3 Results of the Automated Functionality Testing..........118

Appendix E – Screen Images 119

E.1 Testing the Functionalities of the Implemented Designs 119

E.2 Demonstrating the Processes of the Architecture..........120

List of Figures

No.	Description	Page No.
Figure 1.1	Some Smart components in a Smart City	2
Figure 1.2	Some applications of WSNs in the infrastructure of a Water Supply System	3
Figure 3.1	Addition operation	23
Figure 3.2	Subtraction operation	23
Figure 3.3	XOR operation	23
Figure 3.4	Transpose operation	24
Figure 3.5	Swap operation	24
Figure 5.1	Top level architecture of the Efficient and Secure Lightweight Protocol	33
Figure 5.2	Architecture of the Module 'Generate Dynamic Binary Key'	34
Figure 5.3	Architecture of the Module 'Encrypt Plain Data'	37
Figure 5.4	Architecture of the Module 'Decrypt Cipher Data'	38
Figure 7.1	Execution time vs PRNGs for the input data size 25Kbyte	52
Figure 7.2	Execution time vs PRNGs for the input data size 30Kbyte	52
Figure 7.3	Execution time vs PRNGs for the input data size 35Kbyte	53
Figure 7.4	Execution time vs PRNGs for the input data size 40Kbyte	53
Figure 7.5	Execution time vs PRNGs for the input data size 45Kbyte	54
Figure 7.6	Execution time vs PRNGs for the input data size 50Kbyte	54
Figure 7.7	Execution time vs PRNGs for the input data size 55Kbyte	55
Figure 7.8	Execution times of LCGSheffield vs ParkMiller PRNGs architectures	56
Figure 7.9	Execution time vs Hash Algorithms for the input data size 25Kbyte	58
Figure 7.10	Execution time vs Hash Algorithms for the input data size 30Kbyte	58
Figure 7.11	Execution time vs Hash Algorithms for the input data size 35Kbyte	59

Figure 7.12	Execution time vs Hash Algorithms for the input data size 40Kbyte	59
Figure 7.13	Execution time vs Hash Algorithms for the input data size 45Kbyte	60
Figure 7.14	Execution time vs Hash Algorithms for the input data size 50Kbyte	60
Figure 7.15	Execution time vs Hash Algorithms for the input data size 55Kbyte	61
Figure A.1	Architecture of the Module 'Encrypt Data Block: Round 1'	68
Figure A.2	Architecture of the Module 'Encrypt Data Block: Round 2'	68
Figure A.3	Architecture of the Module 'Encrypt Data Block: Round 3'	69
Figure A.4	Architecture of the Module 'Encrypt Data Block: Round 4'	69
Figure A.5	Architecture of the Module 'Decrypt Data Block: Round 1'	70
Figure A.6	Architecture of the Module 'Decrypt Data Block: Round 2'	70
Figure A.7	Architecture of the Module 'Decrypt Data Block: Round 3'	71
Figure A.8	Architecture of the Module 'Decrypt Data Block: Round 4'	71
Figure E.1	Successive screen shots of testing the functionality of the designs with different PRNGs	122
Figure E.2	Successive screen shots of the demonstration processes of the architecture.	127

List of Tables

No.	Description	Page No.
Table 2.1	Summarization of the identified issues in the literature review.	10
Table 5.1	Values of the constants of well-accepted PRNGs	38
Table 5.2	Values of the constants of Selected PRNGs	39
Table 5.3	Properties of the selected Hash Algorithms	
Table 7.1	Measured average execution times for different sizes of input data, with the same hash algorithm.	49
Table 7.2	Efficiency comparison architectures having LCGSheffield and ParkMillereen	54
Table 7.3	Measured average execution times for different sizes of input data, with the same PRNG.	55
Table C.1	Measured execution times of the input data 25 Kbyte - same PRNG	114
Table C.2	Measured execution times of the input data 30 Kbyte - same PRNG	115
Table C.3	Measured execution times of the input data 35 Kbyte - same PRNG	115
Table C.4	Measured execution times of the input data 40 Kbyte - same PRNG	116
Table C.5	Measured execution times of the input data 45 Kbyte - same PRNG	117
Table C.6	Measured execution times of the input data 50Kbyte - same PRNG.	117
Table C.7	Measured execution times of the input data 55Kbyte - same PRNG.	118
Table C.8	Measured execution times of the input data 25Kbyte - same Hash Algorithm	119
Table C.9	Measured execution times of the input data 30Kbyte - same Hash Algorithm.	119

Table C.10	Measured execution times of the input data 35Kbyte - same Hash Algorithm.	120
Table C.11	Measured execution times of the input data 40Kbyte - same Hash Algorithm.	121
Table C.12	Measured execution times of the input data 45Kbyte - same Hash Algorithm.	121
Table C.13	Measured execution times of the input data 50Kbyte - same Hash Algorithm.	122
Table C.14	Measured execution times of the input data 55Kbyte - same Hash Algorithm.	122
Table D.1	List of cipher texts and decrypted plain texts generated by the protocol which have different PRNGs in its architecture	125
Table D.2	List of cipher texts and decrypted plain texts generated by the protocols which have different the Hash Algorithms in its architecture	127
Table D.3	Test results of the Automated Functionality Testing	128

Abbreviations

µTESLA	Micro version of Timed Efficient Streamed Loss-tolerant Authentication
ANSI	American National Standards Institute
IDS	Intrusion Detection System
IoT	Internet of Things
LCG	Linear Congruential Generator
LiSP	Lightweight Security Protocol
LLSP	Link Layer Security Protocol
LSec	Lightweight Security Protocol
MAC	Message Authentication Code
MEMS	Micro Electro Mechanical Systems
NIST	National Institute of Standards and Technology
NSA	National Security Agency
PRNG	Pseudo Random Number Generator
SHA-1	Secure Hash Algorithm -1
SNEP	Sensor Network Encryption Protocol
WSN	Wireless Sensor Network

Chapter 1

Introduction

1.1 Prolegomena

Water supply and sanitation services are required to ensure the uninterrupted satisfactorily services in quantity and quality wise to the growing number of consumers. Traditional non-intelligent distribution infrastructures obstacles to improve the services provided to consumers in an efficient way. Introducing intelligence distribution infrastructure can solve these issues in a very cost effective manner. Already, some countries have been commenced to implement the intelligent infrastructure to provide better services and reduce financial losses[1]–[5]. Wireless Sensor Networks(WSNs) are prime and essential components of the intelligence infrastructure. Nowadays, WSNs have emerged as modern day technology under the push of recent technological advances in Micro Electro Mechanical Systems (MEMS) technology, wireless communications and digital electronics[6].

Successful implementations of WSNs are influenced by their efficient and secure data communications. Due to the specific features of WSN, traditional security solutions are not suitable for it. This issue creates major challenges of its wide range of applications effectively. Thus, the issue enables the emergence of new dimensions of research to propose suitable efficient and secure solutions[7]. We have conducted a research to offer an efficient and secure lightweight protocol for WSNs having large number of sensor nodes. Our solution has recorded positive performance in efficiency, while maintaining the recommended security features of WSNs.

1.2 Background and Motivation

Sri Lanka is one of the developing countries which quickly adopt modern Information and Communication Technologies(ICT)[8]. Government of Sri Lanka shows much interest to develop major cities as Smart Cities to provide better services to its citizens [9], [10]. Already, this concept has been commenced to implement in major cities of many developed countries to improve the services provided to their citizens in a cost effective manner[1]–[5]. In general, Smart City have several Smart components

including Smart Water Supply & Sanitation services to provide high level services to the people who lived in that city. Improving utilities' performance is crucial to ensure continuous high quality service and lower levels of leakage which affect both the quality and quantity of water available to end-users, and the utilities' revenue and its financial sustainability.

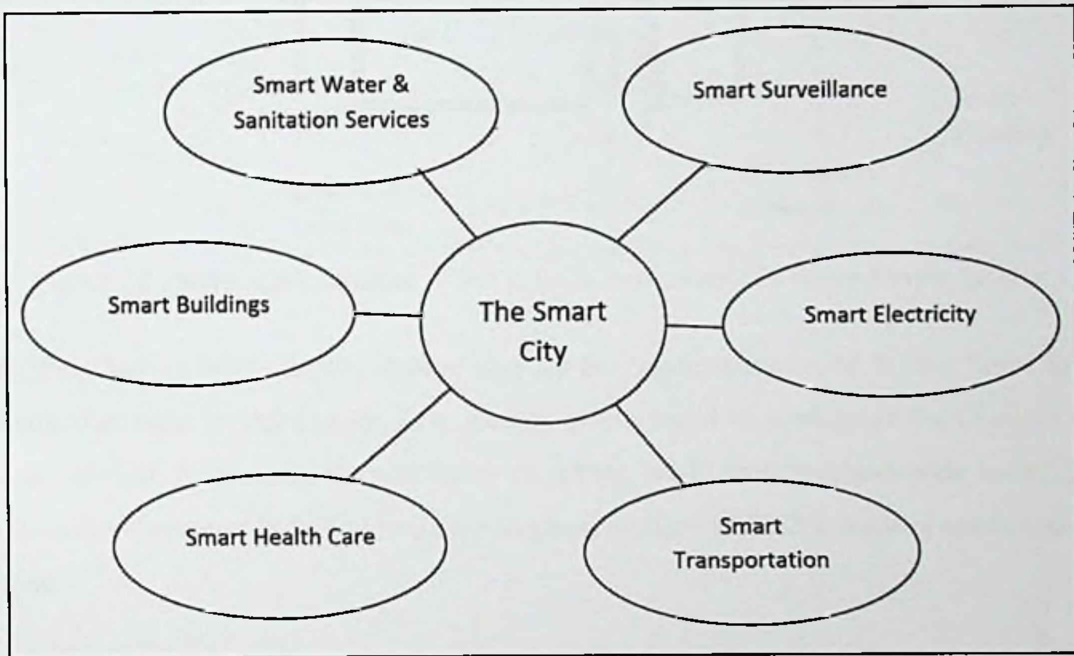


Figure 1.1 : Some Smart components in a Smart City

Smart Water Supply & Sanitation services must have intelligent treatment and distribution systems. Such important intelligent systems require WSNs to monitor the environment efficiently and effectively in a very cost effective manner. WSNs provide a capable platform for low cost, high performance and real-time monitoring. Many researches have been done to utilize WSNs in Water Supply & Sanitation services[11]–[15].

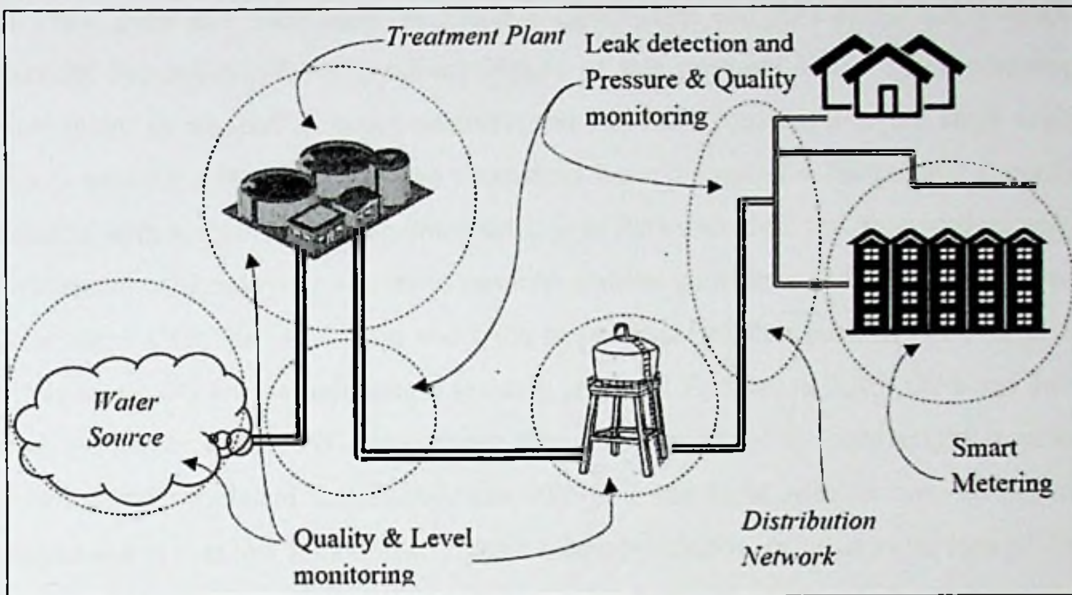


Figure 1.2 : Some applications of WSNs in the infrastructure of a Water Supply System

Further, having Wi-Fi facility, makes easy for the implementation of WSNs. Since Sri Lanka has been selected as the first country in the world to implement the Goolgle's Loon project, It increases the possibility of getting Wi-Fi facility island-wide soon[8]. These developments in ICT upsurge the implementations of WSNs in many application areas.

But, one of the major factors of the successful implementation of WSNs depends on its secure data communication. Traditional security mechanisms cannot be employed on WSNs due its specific features such as resource limitations, computational constraints and deployment in unattended hostile environments[16]. Therefore, specific security mechanisms satisfying WSN's features must be developed. This is a very challengeable task because just increasing the security mechanism will require more computing power and resulting make the mechanism not suitable for WSNs. Similar manner, just modifying the mechanism to consume less computing power results affect the security of the WSNs.

In this study, critical review of wide ranging literatures related to various security threats, security protocols with it evolution and key management schemes of WSNs have been done. Literatures regarding the pseudo random number generators(PRNGs) relevant to these security implementations are also reviewed.

In 2004, Park and Shin have proposed a lightweight protocol called Lightweight Security Protocol(LiSP). The salient feature of this protocol is the novel rekeying mechanism to tradeoff between security and resource consumption for large scale sensor networks. In 2006, Sun and coworkers have presented a lightweight security protocol with similar key management scheme of Park and Shin, but improved security mechanism by employing a pseudo random number generator - Linear Congruential Generator(LCG). In 2015, Jain and Ojha have identified that Park-Miller PRNG is better than LCG for the lightweight security protocol. Further, in 2015, Ojha and Jain analysed some other PRNGs to evaluate the performances of the lightweight security protocol and concluded that Park-Miller PRNG is the most suitable one. But these studies didn't consider Park-Miller's latest recommendation, or other variations of the PRNG. PRNGs play vital role in the security and efficiency of the lightweight protocol[17]. Moreover, It has been identified that Secure Hash Algorithm-1 employed in this protocol has similar effect as PRNG in the security and efficiency of the protocol.

Therefore, performance of the lightweight protocol can be enhanced without compromising its security features, by utilizing more appropriate PRNG and secure hash functions in its architecture.

1.3 Problem definition

Based on the literature review, it has been identified that there is a lack in the selection of appropriate PRNG and secure hash function when designing the secure lightweight protocol for Wireless Sensor Networks which have large number of sensor nodes and are used to monitor environmental parameters in real-time.

1.4 Aim and Objectives

Aim of the project is to design and develop an efficient and secure lightweight protocol for WSNs which have large number of sensors deployed in hostile environments.

There are two types of objectives have been identified for this study such as related to the problem and related to the solution.

1.4.1 Objectives related to the problem

- Critically review the developments and issues in the secure lightweight protocols of WSNs which have large number of sensor nodes and are used to monitor environmental parameters in real-time.
- Critically study the technology used for designing and implementing the secure lightweight protocols for WSNs which have large number of sensor nodes and are used to monitor environmental parameters in real-time.

1.4.2 Objectives related to the solution

- Identify the variables which can affect the efficiency of the secure lightweight protocol.
- Search and identify suitable new values for the identified variables.
- Design secure lightweight protocols with the selected new values, in addition to the previously proposed value for bench mark.
- Learn the technology, MATLAB, which is most suitable for the implementation and evaluating the selected values.
- Develop prototype models using MATLAB for the designed protocols.
- Test the functionalities of the implemented prototype models.
- Measure the computation times of the implemented models for various sizes of input data.
- Analyse the results obtained during the measurement of computation times.
- Interpret the analysis, conclude a decision and recommended it.
- Identify appropriate PRNGs including the PRNGs used in the study[18] as control parameter.
- Prepare the documentation for the project work.

1.5 Hypothesis

Performance of a secure lightweight protocol of WSNs can be increased without compromising any of its security features, by utilizing a suitable PRNG and a secure hash algorithms in its architecture.

1.6 Resources required

Resources required for the design and implementation is a personal computer with MS Windows Operating System, MATLAB, and MS Excel.

Resources required to measure the computation time are two virtual machines with identical configurations with MS Windows Operating System, MATLAB, and MS Excel on a server which must be capable to function long hours without disturbances.

1.7 Structure of the thesis

The rest of the thesis is structured as follows. Chapter 2 is on critical review of the area of security of WSNs and PRNGs relevant of the security of WSNs. Chapter 3 presents technology adopted toward an efficient and secure lightweight protocol for WSNs. Chapter 4 provides the overall picture of our novel approach to design the lightweight protocol. Chapter 5 discusses the design of the solution. Chapter 6 is about the implementation of the secure light weight protocol. Chapter 7 reports on the evaluation of the proposed solution. Chapter 8 concludes the results analysed in the evaluation with the note for further work.

1.8 Summary

This chapter gave the overview of the thesis. Importance, background and motivation of this project have been discussed initially. Studies done in this domain also presented briefly. Problem definition, Objectives, Hypothesis, and Resources required to perform the project also expressed. Finally, structure of thesis described chapter by chapter. Next chapter provide details regarding the developments in security of WSNs and PRNGs relevant to the security of WSNs.

Chapter 2

Developments in security of Wireless Sensor Networks

2.1 Introduction

Chapter 1 has presented the importance and challenges in the security measures of the Wireless Sensor Networks (WSNs). Moreover, it has described the objectives, hypothesis and required resources of the project. This chapter critically reviews wide ranging literature regarding the developments in security protocols of WSNs. Also, literature regarding the pseudo random number generators (PRNGs) relevant to these security implementations is reviewed. At the end of the chapter, summarization of the critical reviews, the problem definition derived and identified technologies identified from the review are presented.

2.2 Review of Literature

Perrig and his colleagues[19] worked to identify variety of security threats including physical tampering, node capture and DoS attacks. Further, they highlighted the state of security procedures such as key establishment, robustness to DoS attacks, authentication and secrecy of WSNs. They suggested that these aspects could be achieved by implementing high level security services rather than existing services. But, they didn't provide any high level security services for WSNs.

Further, Brown et al.[20] identified that security algorithms based on public key such as Rivest, Shamir and Adleman (RSA)[21] require in the order of tens of seconds and upto minutes to perform encryption processes in WSNs. They recommended that public key algorithms are not suitable for WSNs. Although, they didn't suggest any suitable key scheme for WSNs.

Moreover, Carman and coworkers[22] shown that public key algorithms take more processing power in contrast with symmetric key algorithms and hash functions in WSNs. However, in this study, suitable symmetric key algorithms or hash functions for WSNs have not been mentioned specifically.

Suitability of symmetric key algorithms on sensor nodes was analysed by Ganesan et al.[23]. They evaluated the feasibility of software implementations of the algorithms such as RC4[24], skipjack[24], International Data Encryption Algorithm(IDEA)[24] and Advanced Encryption Standard(AES)[24] in sensor nodes. It was concluded that security algorithms such as IDEA and AES give more overhead than others. But, any proposal for the improvements in the less overhead symmetric key algorithms was not made.

In the study by Ahmed[7], five popular security protocols of WSNs such as TinySec[25], LLSP[26], SPINS[27], LiSP[28], and LEDS[29] have been examined. Major features of the selected protocols have been evaluated based on packet transfer types, key management, location-aware, In-Network process and scalability. Also, this study has evaluated the strength and weakness of the protocols based on the security requirements such as Encryption, Authentication, Message Integrity, Availability and Secure localization. This evaluation supports the selection of suitable security protocol for a given application. Nevertheless, this work doesn't provide any suggestions for improving any of the security protocols, or propose any new security protocol for WSNs.

Moreover, Ren et. al.[29] have proposed a location aware end-to-end security protocol called as LEDS. It provides en-route filtering, end-to-end authentication and location aware key management. It can be used in small as well as large WSNs. The drawback in this study is the number of keys utilized in the cryptosystem are proportional to the cell size. In addition, this protocol doesn't prop up dynamic topology which is required by most of the WSNs.

Further, Karlof and coworkers have proposed a link layer lightweight security protocol for WSNs, named as TinySec[25]. This protocol supports two special security options such as authenticated encryption (TinySecAE) and authentication only (TinySecAuth). Also, It is included into the official TinyOS release. The main limitation of this protocol is that It is not fully resistant against node capture attack created by compromised nodes.

An energy efficient link-layer security protocol(LLSP) for WSNs has been developed by Jian et. al.[30]. This protocol has been developed based on the idea of TinySec. But,

dissimilar packet format and crypto structure are used by this protocol. It ensures message confidentiality, message authentication, access control and replay protection. The major drawback of this protocol is low scalability as node counters are unable to maintain large networks.

Furthermore, Perrig and coworkers have presented a link layer security protocol for WSNs named as SPINS[27]. SPINS consists of Micro version of Timed Efficient Streamed Loss-tolerant Authentication(μTESLA), Sensor Network Encryption Protocol(SNEP) and a routing protocol based on these protocols. μTESLA protocol is used in identity authentication broadcasting. SNEP provides semantic security, identity authentication, recursion protection, data freshness and low communication overhead. However, this protocol has scalability issues because the number of required security keys for this protocol are directly proportional to the number of nodes in the network.

Jeffery and coworkers[31] proposed a lightweight security protocol. This protocol functions in the base station. It provides security by enabling the base station to detect and remove the compromised nodes in WSNs. This lightweight security protocol doesn't specify any security measures for the protection against the passive attacks on a node where an adversary is intercepting the communication. Moreover, this scheme increases the overhead of neighbouring nodes of the base station.

Later, Park and Shin[28] have proposed another lightweight security protocol for WSNs called Lightweight Security Protocol (LiSP). The salient feature of this protocol is the novel rekeying mechanism to tradeoff between security and resource consumption for large scale sensor network. It uses a stream cipher for its cheap and fast processing. The active temporal key stream is directly applied to the input stream for the encryption process. The main constraints of the proposed protocol are usage of stream cipher rather than block cipher[24], limited number of temporal keys and direct application of the temporal keys as secret keys for encryption/decryption processes.

Further, Shaikh et. al. have presented a lightweight security protocol for distributed WSNs called as LSec[32]. This protocol utilizes energy and memory efficient technique that assumes grouping network nodes into clusters. It is designed with symmetric and asymmetric security schemes. Base station must be trusted party in this scheme because

it needs to access the public keys of all nodes in the network. However, this protocol suffers from higher communication overhead due to neighbouring nodes of base station by forwarding requests and response packets during authentication and authorization phases.

Sun and coworkers[16] have worked to improve the security protocol of Wireless Sensor Network - LiSP[28]. Advance feature of this proposal is the usage of block cipher [24] and Pseudo Random Number Generator(PRNG), Linear Congruential Generator(LCG) [24] to generate pseudo random numbers in the cryptosystem. Performance analysis proved that the proposed algorithm is more efficient than RC5[24]. The main restrictions in this proposed protocol is the usage of the random numbers generated directly as secret keys in the cryptosystem. Moreover, this study has not analysed other suitable PRNGs to evaluate the performance of the algorithm.

Moreover, Mohamood et al.[33] have presented a hybridize dynamic symmetric key cryptography using LCG algorithm. It deals with a dynamic symmetric key cryptographic method[24] using substitution and transposition techniques. In this algorithm, dynamic secret key has been generated using LCG[24], user input key, built-in key and SHA-1[34] hashing scheme. The main advantage of this algorithm is the creation of a new secret key for every pair of encryption and decryption processes. This proposed algorithm has been demonstrated the best performance when compared with other popular cryptography algorithms such as DES[34], 3DES[34], Rijndael[35] and Blowfish[36]. The main drawback of this study is that it has not analysed other suitable PRNGs to evaluate the performance of the algorithm.

Further, Mahmood et al.[37] have presented a symmetric key cryptographic algorithm using dynamic key. In this algorithm, LCG, user input key, built-in key and hash function SHA-1[34] have been used to generate dynamic secret keys for encryption and decryption processes. This algorithm employs four rounds of encryption and decryption processes with different parts of the dynamic secret keys. The block cipher method has been implemented in which input data are divided into blocks. This proposed algorithm has been demonstrated the best performance when compared with other popular cryptography algorithms such as DES[34], 3DES[34], Rijndael[35] and Blowfish[36]. Moreover, this algorithm is more efficient and secure than the algorithm

proposed in [33]. The main shortcomings in this study is that it has not analysed other suitable PRNGs to evaluate the performance of the algorithm.

Jain and Ojha[18] have extended the study done in [16], [33], and [37]. Park-Miller's PRNG[38] algorithm has been used to generate random numbers rather than LCG as implemented in [16], [33] and [37]. Four rounds of encryption and decryption process has been implemented as similar in the study [37]. One of the strong security features of this work is the use of dynamic key to encrypt and decrypt messages. Block cipher algorithm[39] has been used to encrypt and decrypt the data. This research has shown that Park-Miller algorithm is more efficient than the LCG algorithm[34] to design the protocol. This study doesn't consider the latest recommendations made by Park-Miller regarding their proposed algorithm. Also, It has not analysed other suitable PRNGs, except LCG and Park-Miller, to evaluate the performance of the security protocol.

Jain and Ojha have extended the previous study [18] by using different PRNG generator algorithms to identify the efficient PRNG algorithm to implement LiSP[40]. PRNG algorithms such as LCG[34], RC4[41], Park-Miller[38], Blum Blum shub[42], and Wichman-Hill[43] have been tested and time and energy taken when using these algorithms have been measured. Eventually, It has been found that Park-Miller's PRNG is the best one for the implementation of the security protocol - LiSP. The main limitation of this study is that it has not analysed other suitable PRNGs to evaluate the performance of the security protocol. This study also doesn't consider the latest recommendations made by Park-Miller regarding their proposed algorithm.

Park and Miller[38] have proposed a new algorithm to generate pseudo random numbers based on Lehmer PRNG[44]. This study has recommended two different implementations for integer calculation and real calculation. This scheme has good features such as full period sequence, efficient implementation with 32-bit arithmetic, and satisfactorily random sequence. However, the proposed implementations are not suitable for low level languages.

Later, Carta[45] has optimized the implementation of the PRNG suggested by Park and Miller[38]. In this implementation, the algorithm for calculations of Park and Miller PRNG has been simplified by introducing shift operations. Therefore, It is more

suitable for low level languages. But, the efficiency of the algorithm has not been demonstrated experimentally with lightweight security protocols.

Further, Marsaglia[43] proposed a variant of LCG. The Box-Muller algorithm[46] has been improved by Marsaglia in a way that the use of trigonometric functions can be avoided. It is important, since the computation of trigonometric functions is very time-consuming. So, it has been theoretically proved that efficient than Box-Muller algorithm. However, the efficiency of the algorithm has not been demonstrated experimentally with lightweight security protocols.

2.3 Summarization of the reviews and Problem definition

The above study shows numerous limitations of the security protocol of WSNs. Among other issues, methods used in cryptography algorithms and generation of dynamic keys relevant to the cryptography can be highlighted. These issues are summarized in Table 2.1

	Research	Limitations
1	"Security in wireless sensor networks" by Perrig, 2004 [1].	Didn't provide any high level security services for WSNs.
2	"PGP in Constrained Wireless Devices" by Brown et al. 2000 [20].	Didn't suggest any suitable key schemes for WSNs.
3	"Constraints and approaches for distributed sensor network security" by Carman et al. 2000 [22].	Suitable symmetric key algorithms or hash functions for WSNs have not been mentioned particularly.
4	"Analyzing and modeling encryption overhead for sensor network nodes" by Ganesan et al. 2003 [23].	No proposal for the improvements in the low overhead symmetric key algorithms.
5	"An Evaluation of Security Protocols on Wireless Sensor Network" by Ahmed2009 [7].	No suggestions for improving the existing security protocols or proposal for new security protocol for WSNs.
6	"LEDS: Providing Location-aware End-to-end Data Security in	Number of keys used is limited by the cell size

	Wireless Sensor Networks." By Ren et. al. 2006 [29]	and Doesn't prop up dynamic topology.
7	TinySec: a link layer security architecture for wireless sensor networks by Karlof et al 2004[25].	It is not fully resistant against node capture attack created by compromised nodes.
8	"A Power Efficient Link-Layer Security Protocol (LLSP) for Wireless Sensor Networks" by Jian et. al. 2005.	Low scalability as node counters are unable to maintain large networks.
9	"SPINS: Security protocols for sensor networks" by Perrig et al. 2002 [27].	Has scalability issues because number of security keys are proportional to number of nodes in the WSN. Therefore, not suitable for larger WSNs.
10	"Security for sensor networks" by Undercoffer et al. 2002 [31].	No protection against passive attacks, and Increases the overhead of neighbouring nodes of the base station.
11	"LiSP: A lightweight security protocol for wireless sensor networks" by Park and Shin 2004[28].	The usage of stream cipher, limited number of temporal keys and direct application of temporal keys for encryption are the main drawbacks.
12	"LSec: lightweight security protocol for distributed wireless sensor network" by Shaikh et al. 2006 [32].	Base station must be trusted node and It suffers from higher communication overhead.
13	"A lightweight secure protocol for wireless sensor networks" by Sun et al. 2006 [16].	The usage of the random numbers generated directly as secret keys in the cryptosystem and Considered only one PRNG for generating dynamic keys.
14	"Hybridize Dynamic Symmetric Key Cryptography using LCG" by Mohamood et al. 2012 [33].	Considered only one PRNG to generate dynamic keys. Has only two rounds of encryption. Also, less efficient and secure than the algorithm proposed in the study[37]

15	"Symmetric Key Cryptography using Dynamic Key and Linear Congruential Generator (LCG)" by Mohamood et al. 2012 [37].	Considered only one PRNG to generate dynamic keys.
16	"Implementation of LiSP using Park-Miller for Wireless Sensor Network" by Jain and Ojha 2015 [18].	Not analysed other suitable PRNGs, except LCG and Park-Miller, to evaluate the performance, Not considered latest recommendations made by Park-Miller regarding their proposed PRNG, and Not considered latest recommendations made by Park-Miller regarding their proposed PRNG.
17	"Implementation of LiSP using Different Random Number Generator as a Dynamic Key for Wireless Sensor Network" by Ojha and Jain 2015 [40].	Not considered other available simple PRNGs for generating dynamic keys and Not considered latest recommendations made by Park-Miller regarding their proposed PRNG.
18	"Random number generators: good ones are hard to find" by Park and Miller 1988 [38].	Not suitable for low level languages.
19	Two fast implementations of the 'minimal standard' random number generator", by Carta 1990[45].	Efficiency of the algorithm has not been demonstrated experimentally with lightweight security protocols.
20	"A comparison of four pseudo random number generators implemented in Ada" by Graham 1992[43] .	Efficiency of the algorithm has not been demonstrated experimentally with lightweight security protocols.

Table 2.1 : Summarization of the identified issues in the literature review

2.3.1 Problem Definition

Based on the above critical review, the research problem is defined as the inadequate attention given to the algorithms of pseudo random number generators(PRNGs), when designing and developing secure lightweight protocols for WSNs.

2.3.2 Identified Technologies

Some technologies have been identified as more suitable for designing and developing lightweight security protocols for WSNs by reviewing the literature. The technologies are Symmetric key cryptosystem, One Time Pad(OTP), Secure Hash Algorithm(SHA), Block cipher cryptosystem, Dynamic key and Group-based key management. These technologies and their relevant our proposed solution will be described in the next chapter.

2.4 Summary

This chapter discussed the studies done in the areas of security of WSNs and PRNGs relevant to the design and implementation of the lightweight security protocols of WSNs. Protocols proposed to implement the security mechanism on WSNs including TinySec, LLSP, LEDS, SPINS, LiSP and LSec have been critically analysed. Further, problem definition has been derived by analyzing the critical reviews of these literatures. Identified technologies in the literature have also been presented. Next chapter, technologies adopted for designing and developing the secure lightweight protocol will be discussed. These technologies have been identified as suitable technologies in designing and developing the lightweight security protocols for WSNs.

Chapter 3

Technologies adopted for the proposed protocol

3.1 Introduction

Chapter 2 discussed the studies have been done in the security protocols for wireless sensor networks and pseudo random number generators. Some suitable technologies for the design and implementation of the proposed protocol have been identified in the literature reviews. Those technologies are Dynamic key, Onetime pad(OTP), Pseudo Random Number Generators(PRNGs), Block cipher, Hash function, Symmetric key, Group-based key management, and Encryption primitives. Further, MATLAB application, as used in many studies found in the literature review, is chosen for modeling and evaluating the proposed algorithm. The identified technologies are described in this chapter.

3.2 Onetime pad(OTP)

A onetime pad(OTP) is an encryption technique in which a key generated randomly is used only once to encrypt a message. Then, the key will be discarded to avoid the future usage of the key for encryptions. Messages encrypted with keys based on randomness have the advantage that there is theoretically no way to "break the code" by analyzing a succession of messages. Each encryption is unique and bears no relation to the next encryption. During the decryption, the same key used to encrypt the message must be required. This situation raises the problem of how to get the key for decryption which is at different location safely or how to keep the keys securely at both encryption and decryption locations. The key used in a one-time OTP is called a secret key because if it is revealed, the messages encrypted with it can easily be deciphered[37], [40], [18], [33].

Further, if the secret key is truly random, is at least as long as the plain text, is never reused in whole or in part, and is kept completely secret, then the resulting cipher text will be impossible to decrypt or break[24].

This technique has been implemented in the proposed secure lightweight protocol to increase the security. PRNG is used to ensure the creation different keys at each encryption process.

3.3 Pseudo Random Number Generator (PRNG)

Random numbers are critical in every aspect of cryptography. Unfortunately, true random numbers are very difficult to generate, especially on computers that are typically designed to be deterministic. This brings the concept of pseudo-random numbers, which are numbers generated from some random internal values, and that are very hard for an observer to distinguish from true random numbers. It is important to see the difference between the meaning of pseudo-random numbers in the context of cryptography, where they must be indistinguishable from real random numbers.

A pseudo random number generator (PRNG) is an algorithm for generating a sequence of numbers whose properties approximate the properties of sequences of random numbers. The PRNG-generated sequence of numbers is not truly random, because it completely determined by a relatively small set of initial values and parameters of the mathematical equation. Good statistical properties and time efficient are a central requirement for the output of a PRNG. PRNGs are important in practice for their speed in number generation and their reproducibility[43], [42], [47].

PRNG is a basic for the generation of dynamic key of the proposed secure light weight security protocol. The below mentioned PRNGs are implemented with the protocol to evaluate and choose an appropriate PRNG.

3.3.1 Linear Congruential Generator (LCG)

A linear congruential generator (LCG) is an algorithm that yields a sequence of pseudo randomized numbers calculated with a discontinuous piecewise linear equation. The method represents one of the oldest and best known pseudo random generator algorithms. This method can be easily implemented and fast, especially on computer hardware which can provide modulo arithmetic by storage-bit truncation[42].

The generator is defined by the recurrence relation:

$$X_{n+1}=(aX_n+c) \bmod m$$

Where X is the sequence of pseudo random values, and

m, $0 < m$ – the 'modulus'

a, $0 < a < m$ – the 'multiplier'

c, $0 \leq c < m$ – the 'increment'

X_0, $0 \leq X_0 < m$ – the 'seed' or the 'start value'

are integer constants that specify the generator.

Algorithm of LCG to generate pseudo random number is given below.

function LCGPseudoRandomNumber : real

// Assume seed is global variable and has an initial value.

Step 1: Initialize parameters

a ←48271; m←2147483647; c←4;

Stept 2: X1← (a x seed + c) mod m

Step 3: seed ← X1

Step 4: return ← y1/m

3.3.2 Park-Miller pseudo random number generator

Park and Miller[38] proposed minimal standard for LCG and efficient way for implementation.

The generator is defined by the recurrence relation:

$$X_{n+1}= aX_n \bmod m$$

Where X is the sequence of pseudo random values, and

m =2147483647 – the 'modulus'

a =16807 – the 'multiplier'

X_0, $0 \leq X_0 < m$ – the 'seed' or the 'start value'

are integer constants that specify the generator.

LIBRARY
UNIVERSITY OF MORATUWA, SRI LANKA
MORATUWA

Algorithm of ParkMiller to generate pseudo random number is given below.

function ParkMillerPseudoRandomNumber : real

// Assume seed is global variable and has an initial value.

Step 1: Initialize parameters

a ←16807; m←2147483647; q←127773; r←2836;

Step 2: hi ← seed div q

Step 3: lo ← seed mod q

Step 4: test ← a x lo – r x hi

Step 5: if (test>0) then seed ← test

else seed ← test + m

Step 6: return ← seed /m

3.3.3 Park-Miller-Carta pseudo random number generator

Carta[45] has improved the PRNG implementation proposed by Park and Miller[38].

Algorithm of ParkMillerCarta to generate pseudo random number is given below.

function ParkMillerCartaPseudoRandomNumber : real

// Assume seed is global variable and has an initial value.

Step 1: Initialize parameters

a ←16807; m←2147483647;

Step 2: lo ← (a x (seed & 0xFFFF))

Step 3: hi ← (a x (seed >> 16)

Step 4: lo ← lo + (hi & 0x7FFF) <<16

Step 5:lo ← lo + (hi >> 15)

TH 3168

Step 6: if (lo > 0x7FFFFFFF) then lo ← lo - 0x7FFFFFFF

Step 7: seed ← lo

Step 7: return ← lo /m

3.4 Block cipher

In this cryptography technique, messages are divided into fixed length blocks before the encryption process. Then encryption will be performed block by block. In the same way, during the decryption process, the cipher data is divided into fixed length blocks and decrypted block by block[24]. In the study[48], It has been recognized that block cipher is more suitable than stream cipher to get higher security in WSNs[47].

3.5 Dynamic key

Cryptosystems utilizes keys to encrypt messages into cipher data and decrypt cipher data into original messages. Cryptosystems utilizes more than one key to strengthen the security of the cipher text. Dynamic key cryptosystems utilizes key which varies on subsequent execution of the processes of the cryptosystems. Security protocols which employ dynamic key cryptosystems are immune to known-key attacks.

Dynamic key means a kind of key which is changed every time. It is new and an advanced concept. These dynamic keys are used for each pair of encryption and decryption. After the decryption these dynamic keys are discarded. Therefore, it is called as dynamic key. Whole keys are not shared between the parties, sender and receiver, but little bit information are shared between two parties. So on the basic of this information, both parties produce the dynamic key. In this key is applied different part of the message[47], [33], [28], [37]; [24].

3.6 Hash function

One of the fundamental primitives in modern cryptography is the cryptographic hash function, often informally called a one-way hash function. A hash function is the implementation of an algorithm that, given some data as input, will generate a short fixed length result called digest or hash. In another way, it can be defined as a computationally efficient function mapping binary strings of arbitrary length to binary strings of some fixed length, called hash values[24].

Information security often includes situations where a user wants to transform one block of information into another block of information in such a way that the original block cannot be recreated. Also, it is very essential the input block is processed; it will produce the same output block. This means that the process is deterministic.

A cryptographic hash can be described as f(message)=hash and has property that the hash function is one way. A given hash value cannot feasibly be reversed to get a message that produces the hash value. ie. there is no useful inverse hash function f'(hash)=message.

Good hash functions must have the following properties:

Preimage resistant : Given H it should be hard to find M such that H=hash(M).

Second preimage resistant : Given an input m1, it should be hard to find another input, m2 (not equal to m1)such that hash(m1)=hash(m2).

Collision-resistant : It should be hard to find two different messages m1 and m2 such that hash(m1)=hash(m2). Because of the birthday paradox this means the hash function must have a larger image than required for peimage-resistance[24].

3.6.1 Secure Hash Algorithm -1

Secure Hash Algorithm -1 (SHA -1)[49] is in widespread use and was designed to provide protection against collision finding of 2^{80}. It was proposed by National Institute of Standards and Technology(NIST) for certain U.S. federal government applications. The hash value of SHA-1 is 160 bits[24]. It is a member of the Secure Hash Algorithm family which has four algorithms.

3.7 Symmetric key Cryptosystems

Symmetric key Cryptosystems is one that uses the same key to encrypt messages as it does to decrypt messages[37], [28], [18], [24].

Formal definition:

Any cryptosystem on a symmetric key cipher conforms to the following definition:

M : message to be enciphered

K : a secret key

E : enciphering function

D : deciphering function

C : enciphered message. C=:E(M,K)

For all M, C, and K, M=D(C,K) = D(E(M,K), K)

Rehman et al.[48] have analysed many cryptographic techniques for WSNs. They have identified that symmetric key techniques offer better energy efficiency than public key techniques.

This technique is also called as secret-key cipher, or one-key cipher, or private-key cipher, or shared-key cipher.

3.8 Encryption primitives of Cryptosystems

Encryption primitives of cryptosystems are basic operations to encrypt plain text into a cipher data and decrypt cipher data into plaint text. Encryption primitives adopted in this work are Addition operation, Subtraction operation, XOR operation and Transpose operation. These encryption primitives are utilized in the lightweight security protocol[24], [34].

3.8.1 Addition operation

In this operation, the length of the key and block of the message need to be encrypted must have same number of bits. During the operation, key and block of message will be added and the result will be given as output. It is described in the following diagram.

```
Block of message : 0110101111010001
Key              : 0011110111101010 0
                   -------------------------
Cipher data      : 1010100110111010 1
```

Figure 3.1 : Addition operation

3.8.2 Subtraction operation

In this operation, the length of the key and block of the cipher data need to be decrypted must have same number of bits. During the operation, key will be subtracted from the block of cipher data and the result will be given as output. It is described in the following diagram.

```
Block of cipher data : 0110101111010001
Key                  : 00111101111010100
                       -------------------------
Original message     : 0010110111100110 1
```

Figure 3.2 : Sùbtraction operation

3.8.3 XOR operation

In this operation, the length of the key and block of the message or cipher data need to be encrypted or decrypted must have same number of bits. During the operation, XOR operation will be performed between input data and the key and the result will be given as output. It is described in the following diagram.

```
Input data   : 0110101111010000 1
Key          : 0011110111101010 0
               -------------------------
Output data  : 1010110001110101 0
```

Figure 3.3 : XOR operation

3.8.4 Transpose operation

In this operation, the block of message is arranged in square format row by row, then the arranged data will be read column by column to perform encryption. The reverse operation will be performed for decryption operation.

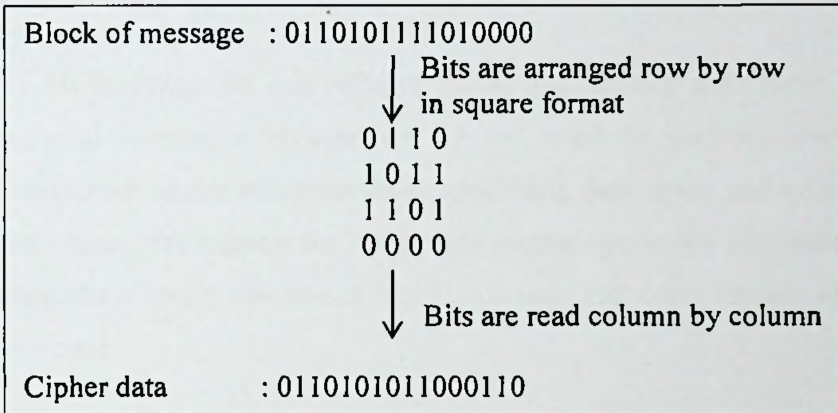


Figure 3.4 : Transpose operation

3.8.5 Swap operation

In this operation, block of message is divided into predefined number of sub blocks. Then, these sub blocks of messages will be swapped and merged together by changing its order of position to create cipher data.

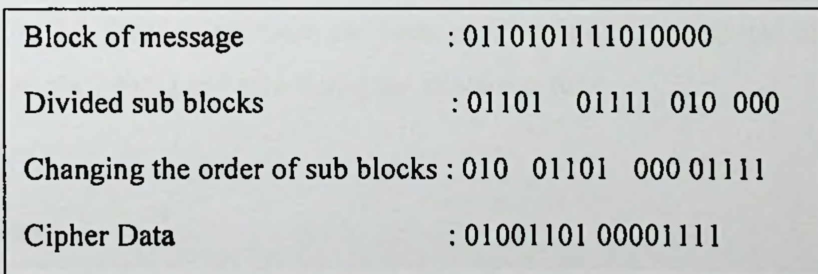


Figure 3.5 : Swap operation

3.9 MATLAB

MATLAB (matrix laboratory) is a multi-paradigm numerical computing environment and fourth-generation programming language. It is proprietary software developed by MathWorks. It allows matrix manipulations, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs written in other languages, including C, C++, Java, Fortran and Python.

MATLAB provides a high-level language and development tools that let us quickly develop and analyze algorithms and applications.

The MATLAB language provides native support for the vector and matrix operations that are fundamental to solving engineering and scientific problems, enabling fast development and execution.

With the MATLAB language, we can write programs and develop algorithms faster than with traditional languages because we do not need to perform low-level administrative tasks such as declaring variables, specifying data types, and allocating memory. In many cases, the support for vector and matrix operations eliminates the need for for-loops. As a result, one line of MATLAB code can often replace several lines of C or C++ code.

We can produce immediate results by interactively executing commands one at a time. This approach lets us quickly explore multiple options and iterate to an optimal solution. We can capture interactive steps as scripts and functions to reuse and automate your work.

MATLAB application has been used for modeling and evaluating the security protocols of WSNs in many studies[40], [18], [33], [37], [17],[50], [51], [52]. Therefore, in this study, MATLAB R2014b software has been used to design, implement and evaluate the proposed algorithms and measuring the execution time.

3.10 Summary

In this chapter, technologies selected to design and implement the proposed security protocol have been described. The mentioned technologies include Dynamic key, Onetime pad(OTP), Pseudo Random Number Generators(PRNGs), Block cipher, Hash function, Symmetric key, Group-based key management, and Encryption primitives. Finally, the MATLAB application used to model and evaluate the algorithm have been described. In the next chapter, approaches pursued to develop the efficient and secure lightweight protocol will be discussed.

Chapter 4

Approach for Efficient and Secure Lightweight Protocol

4.1 Introduction

Chapter 2 presented the critical review of many literatures which were required to design and develop the efficient and secure light weight protocol. Chapter 3 discussed the technology for implementing efficient and secure lightweight protocol for Wireless Sensor Networks(WSNs). This chapter presents our approach to efficiently implementing the lightweight protocol under several headings, namely, hypothesis, input, output, process, users and features. Our approach is lightweight block cipher symmetric key cryptosystem with dynamic key employing onetime pad concept. This chapter highlights the key features that distinguish our novel approach from other existing approaches for security protocols of WSNs. Further, this chapter highlights how the novel approach offers a cost effective and efficient solution for security protocol for WSNs.

4.2 Hypothesis

Performance of a secure lightweight protocol for WSNs can be increased without compromising any of its security features, by utilizing a suitable PRNG and a secure hash algorithm in its architecture.

4.3 Input

When data gathered from environment, Data sensed by wireless sensor nodes are the main inputs for the secure lightweight protocol in the encryption process. These sensed data may include numerical values of temperatures, pressures, PH values, voltages, and mechanical vibrations are main inputs. These inputs are given by the specific sensor nodes deployed in the field. These inputs are given by the sensors continuously in a predefined time intervals. This secure lightweight protocol gets these data in binary forms since it perform processes in bit levels.

When data received from sensor nodes by base stations, encrypted data received by base stations are the main inputs for the secure lightweight protocol in the decryption process. Base stations may receive encrypted data from more than one sensor nodes.

4.4 Output

When data gathered from environment, Encrypted data generated by the secure lightweight protocol are the outputs of the encryption process. Then the data transmitted to the base station directly or via other sensor nodes.

When data received from sensor nodes by base stations, decrypted data generated by the secure lightweight protocol are the outputs of the decryption process. Decrypted data will be in binary form since the secure lightweight protocol perform processes in bit levels. These data may represent the parameters such as temperatures, PH values, pressures, voltages, mechanical vibrations, etc. which have been sensed by the sensors in the field.

4.5 Process

4.5.1 Encryption process

During the encryption process, given binary stream – plain data – will be converted to as cipher data with the involvement of dynamic key. Length of the plain data can be any length which satisfies the hardware configurations of the sensor node. But, length of the plain data and cipher data will be same. Careful consideration is given in the encryption process to increase the security as maximum possible without increasing the computation times.

4.5.1.1 Generation of Dynamic key

Initially dynamic key with the length of 196 bits will be generated. It will be in binary form. Generating random dynamic key requires three basic keys such as built-in key, user input key and pseudo random number. Built-in key is specific and unique for each sensor nodes, that is, each sensor node has a non-identical built-in key and it cannot be modified. User input key will be injected to the sensor nodes before deployment or it will be updated in some predefined long periods. A pseudo random number will be generated using a PRNG during the each encryption processes. Based on these three

keys randomized matrix with the size 14x14 will be generated by performing various matrix operations. Finally, 196 bit stream will be generated as hash value by giving the randomized matrix as input to a hash function.

4.5.1.2 Preparation of data blocks

Before the encryption process, given binary stream – plain data – will be divided into blocks with the size of 49 bits. If the last block doesn't have enough bits for getting 49 bits length, additional pad bits will added to get the required bits.

4.5.1.3 Performing encryption

Using the binary key generated, given plain data will be encrypted block by block. Encryption process has four rounds and each round uses sub keys of 49 bits obtained from the dynamic key for the encryption process. Encryption process includes simple addition, transpose, XOR, and rotation operations. Simple operations are employed because to reduce the computation time and energy in the sensor nodes. Each rounds has applied these basic operations in different manners to increase the security. In the first third and fourth rounds between two basic operations, bits in the block will be swapped in various manners.

After encryption of the all blocks, they will be merged as a single encrypted cipher data.

4.5.2 Decryption process

During the decryption process, given binary stream – cipher data – will be converted to as plain data with the involvement of dynamic key which was used in the encryption process. Length of the cipher data can be any length which satisfies the hardware configurations of the sensor node. But, length of the cipher data and plain data will be same. The decryption process can be viewed as a reverse operation of the encryption process.

4.5.2.1 Generation of Dynamic key

The dynamic key used for the encryption is necessary to decrypt the cipher data. But, in this approach, there is no need to get the dynamic key form the encryption location. The dynamic key with the length of 196 bits can be generated at the decryption location.

It will be in binary form. Generating random dynamic key requires three basic keys such as built-in key, user input key and pseudo random number. Built-in key is specific and unique for each sensor nodes, that is, each sensor node has a non-identical built-in key and it cannot be modified. User input key will be injected to the sensor nodes before deployment or it will be updated in some predefined long periods. A pseudo random number will be generated using a PRNG during the each decryption processes. The pseudo random number must be same the number which was used during the encryption. This can be achieved simply passing the round of the PRNG because the functions of the PRNGs are deterministic. Based on these three keys randomized matrix with the size 14x14 will be generated by performing various matrix operations. Finally, 196 bit stream will be generated as hash value by giving the randomized matrix as input to a hash function. This dynamic key would be same as the dynamic key used for the encryption operation.

4.5.2.2 Preparation of data blocks

Before the decryption process, given binary stream – cipher data – will be divided into blocks with the size of 49 bits. If the last block doesn't have enough bits for getting 49 bits length, additional pad bits will added to get the required bits.

4.5.2.3 Performing decryption

Using the binary key generated, given cipher data will be decrypted block by block. Decryption process has four rounds and each round uses sub keys of 49 bits obtained from the dynamic key for the decryption process. Decryption process also includes simple addition, transpose, XOR, and rotation operations. Simple operations are employed because to reduce the computation time and energy in the sensor nodes. Each rounds has applied these basic operations in different manners. In the first, second and fourth rounds between two basic operations, bits in the block will be swapped in the reverse manner as done in the encryption process.

After decryption of the all blocks, they will be merged as a single decrypted plain data.

4.6 Features

The proposed efficient and secure lightweight protocol has the following features:

- Secret key used for the encryption is not required to share with the decryption party, this feature will reduce the chance of opponents/ eavesdroppers get to know the secret key. Further, it makes easy for the key management process.
- Once a secret key used for encryption will not be used for the successive encryption processes, this will enhance the security by if someone get to know one of the secret key, she/he cannot decrypt successive messages, even if she/he knows the encryption process.
- Utilization of a PRNG which is performed efficiently with the cryptosystem and has good randomness properties. This feature will increase the efficiency of the protocol and ensure excellent randomness of the secret key.
- Utilization of simple encryption primitives in the cryptosystem. This feature increases the efficiency of the protocol.
- Having four rounds of encryption in each of the encryption processes. This feature increases the strength of the cipher data.
- Utilization of different sub keys of the dynamic key at each rounds of encryption. This feature makes impossible to opponents easily decrypt the cipher data.
- Utilization of the symmetric key concept rather than asymmetric key concept in the cryptosystem. This feature increases the efficiency of the protocol.
- Generation of dynamic key based on three basic keys, utilization of matrix operations and utilization of the hash function. This process makes harden to opponents to generate similar key to decrypt the cipher data.
- Utilization of the efficient hash function to generate the binary dynamic key. The feature increase the efficiency of the protocol.
- This protocol supports group wise key management scheme rather than pair wise key management scheme. Therefore, It can be used in large scale WSNs.

4.7 Users

This lightweight protocol will be used by wireless sensor nodes and base stations in the Wireless Sensor Networks(WSNs) to get efficient and secure data communication for their sensitive data. Application areas of the WSNs include smart water supply, structural monitoring, smart metering of utility services, health care, battle field, environmental monitoring, smart traffic controlling, etc.

4.8 Summary

This chapter presented our novel approach to design and develop an efficient and secure lightweight protocol for WSNs. It pointed out how the novel approach offers an efficient and accurate solution for efficient security in WSNs. The next chapter shows the design of the novel approach presented here.

Chapter 5

Design of the Efficient and Secure Lightweight Protocol

5.1 Introduction

Chapter 3 discussed the technology for implementing efficient and secure lightweight protocol for Wireless Sensor Networks(WSNs). Chapter 4 presented the approach to develop an efficient and secure lightweight protocol for WSNs. This chapter elaborates the approach and describes the architecture of the solution. The top level architecture of the solution includes three modules namely Generation of Binary Key, Encryption Process, and Decryption Process. Also, this chapter briefly analysis the theoretical aspects of the architecture and describes the modules which have been used to evaluate the suitable PRNG and Hash function for the proposed protocol.

5.2 Top Level Architecture of the Secure Lightweight Protocol

The top level architecture of the efficient and secure lightweight protocol is shown in Figure 5.1. Within the architecture, modules 'Encrypt Plain Data' and 'Decrypt Cipher Data' work as the core of the solution. Module, 'Generate Dynamic Binary Key', is connected and facilitate the functions of the modules, 'Encrypt Plain Data' and 'Decrypt Cipher Data'. Next we briefly describe the function of each modules.

5.2.1 Generation of Dynamic Binary Key

For the function of the modules, Encryption Process and Decryption Process, 196 bits length binary key is essential. This key has to be renewed during the encryption process of each messages before the message is transmitted. Therefore, every encryption process require a new 196 bits length binary key.

The module, Generation of Dynamic Binary Key, provides 196 bits length dynamic binary key at the encryption process of every messages. Similar manner, during the decryption process, this module, Generation of Dynamic Binary Key, provides 196 bits length dynamic binary key at the decryption process of every cipher messages. This

module requires two basic keys such as user input key and built-in key to generate 196 bits length binary keys. Architecture of this module is shown in Figure 5.2.

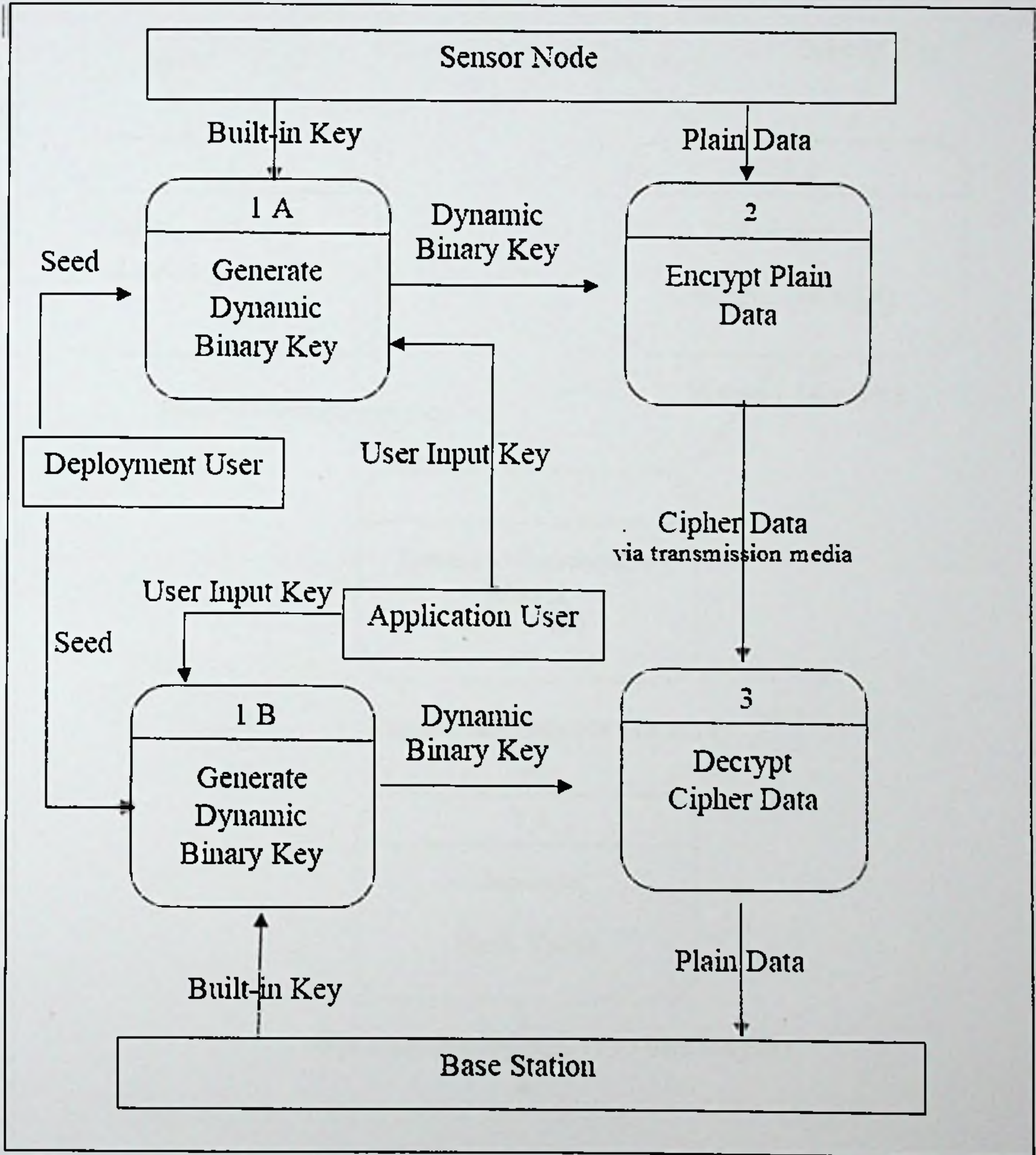


Figure 5.1 – Top level architecture of the Efficient and Secure Lightweight Protocol
(Module "1 A" and "1 B" are identical processes, but in different devices)

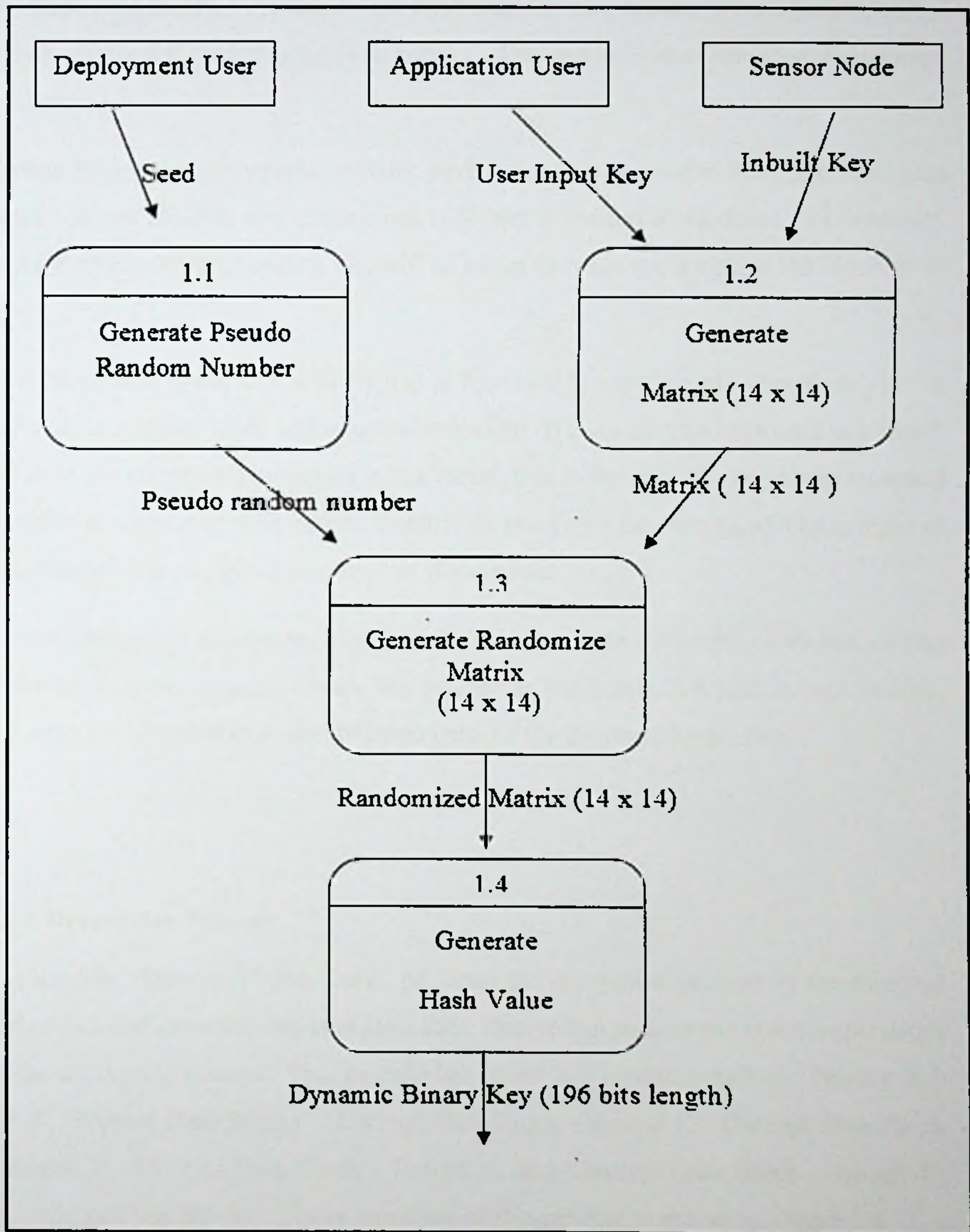


Figure 5.2 – Architecture of the Module 'Generate Dynamic Binary Key'

5.2.2 Encryption Process

The module, 'Encrypt Plain Data', performs the encryption process on the received plain data and gives the output as chipper data. This module has seven sub modules such as 'Prepare Sub Keys', 'Prepare Data Blocks', 'Encrypt Data Block – Round 1',

'Encrypt Data Block – Round 2', 'Encrypt Data Block – Round 3', and 'Encrypt Data Block – Round 4', and 'Merge Data Blocks'. The architecture of this module is shown in Figure 5.3.

During the encryption process, initially, given plain data is divided into fixed sized data blocks. In this scheme, size of the block is 49 bits. If the last block doesn't have enough bits for 49 bits length, padding bits will be added to make the length of the block as 49 bits.

Then, each data block will be encrypted in four rounds using simple operations such as addition, transpose, XOR, and rotation operations. If there are two keys used in a round, between the encryption processes in the round, bits in the data blocks will be swapped in different manner in each rounds. Finally, all encrypted data blocks will be merged as one chipper data and given as output of the module.

For the encryption process in each module, sub keys with the length of 49 bits will be obtained from the dynamic binary key which has the length 196 bits. In each rounds, sub keys are obtained from the different parts of the dynamic binary key.

5.2.3 Decryption Process

The module, 'Decrypt Cipher Data', performs the decryption process on the received cipher data and gives the output as plain data. Decryption process has reverse operations of the encryption process. This module has seven sub modules such as 'Prepare Sub Keys', 'Prepare Data Blocks', 'Decrypt Data Block – Round 1', 'Decrypt Data Block – Round 2', 'Decrypt Data Block – Round 3', and 'Decrypt Data Block – Round 4', and 'Merge Data Blocks'. The architecture of this module is shown in Figure 5.4.

During the decryption process, initially, given plain data is divided into fixed sized data blocks. In this scheme, size of the block is 49 bits. If the last block doesn't have enough bits for 49 bits length, padding bits will be added to make the length of the block as 49 bits.

Then, each data block will be decrypted in four rounds using simple operations such as addition, transpose, XOR, and rotation operations. If there are two keys used in a round,

between the decryption processes in the round, bits in the data blocks will be swapped in different manner in each rounds. Finally, all decrypted data blocks will be merged as one plain data and given as output of the module.

For the decryption process in each module, sub keys with the length of 49 bits will be obtained from the dynamic binary key which has the length 196 bits. In each rounds, sub keys are obtained from the different parts of the dynamic binary key.

Diagrammatical descriptions of the design architectures of the modules 'Encrypt Plain Data' and 'Decrypt Cipher Data' are given in the next pages.

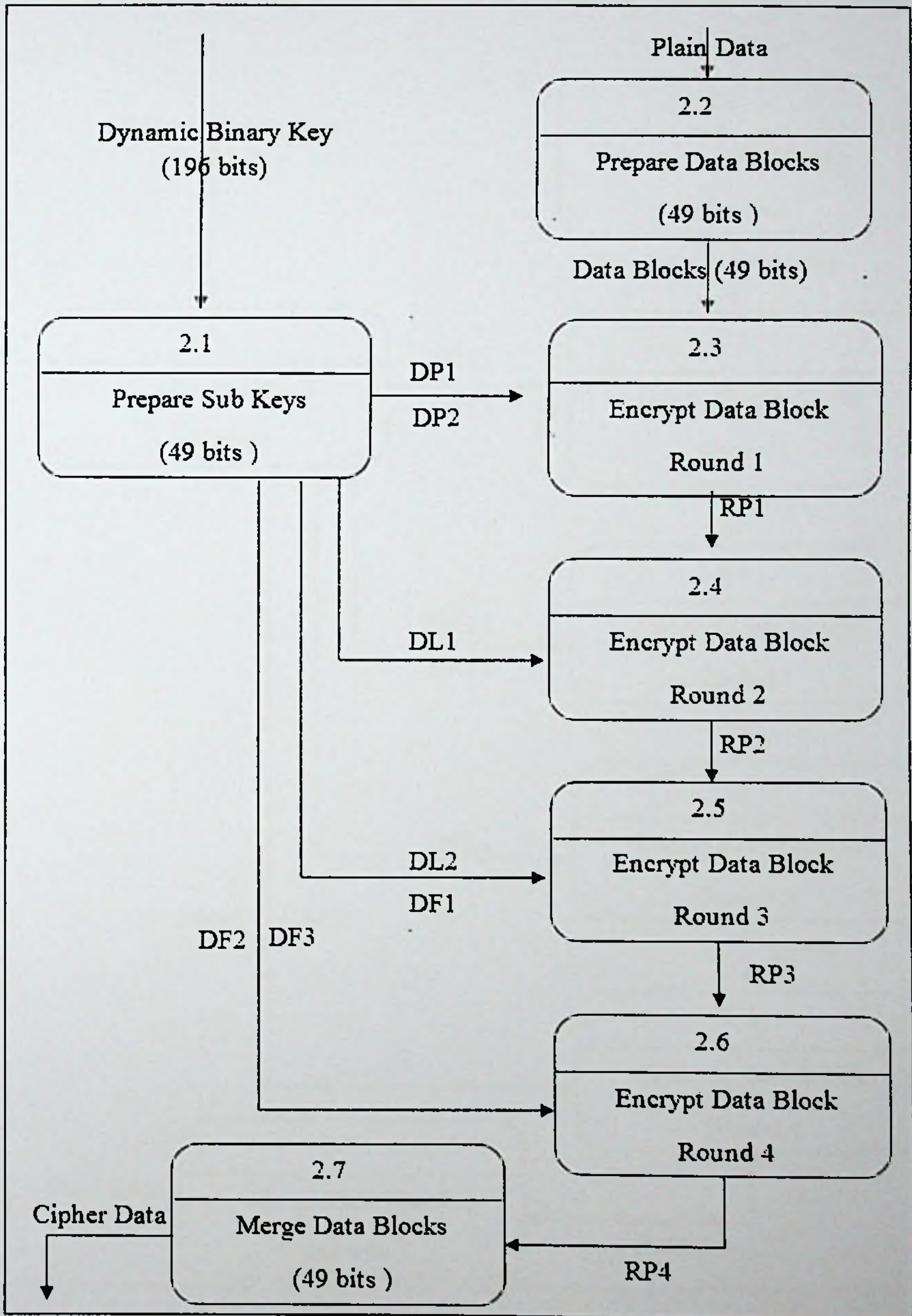


Figure 5.3 – Architecture of the Module 'Encrypt Plain Data'

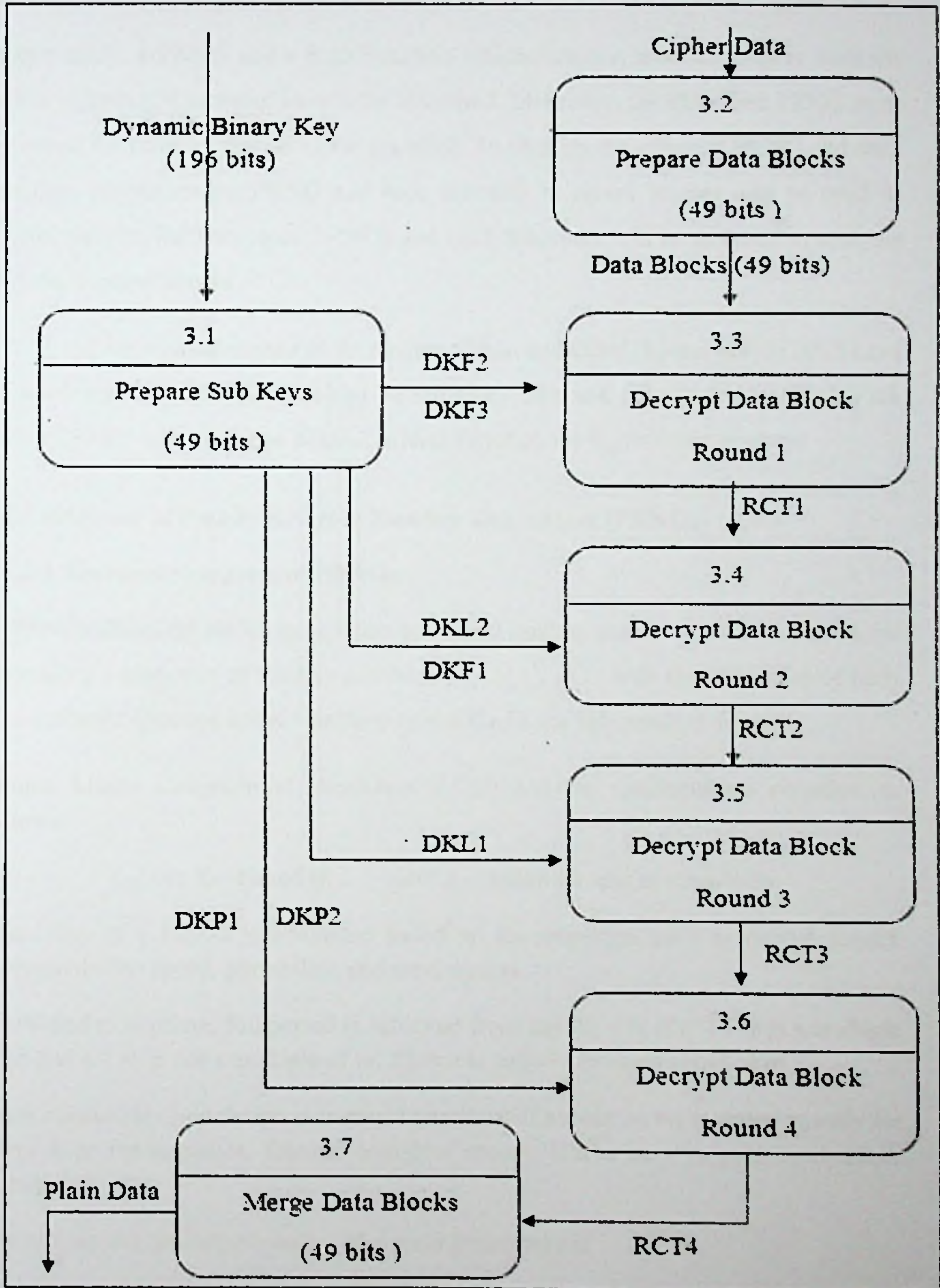


Figure 5.4 – Architecture of the Module 'Decrypt Cipher Data'

5.3 Theoretical Analysis of the Architecture

In this study, a PRNG and a hash function which function most efficiently with the secure lightweight protocol have to be identified. Moreover, the identified PRNG must not affect the security feature of the protocol. To identify the efficient PRNG and hash function, recommended PRNG and hash function in recent studies will be used as control factors. Further, some PRNGs and hash functions will be selected to analysis with the control factors.

PRNG and hash function used in the studies of Jain and Ojha[18] and [40] in 2015 have been selected as controlled variables for our study. Jain and Ojha have identified 'Park Miller PRNG' as PRNG and SHA-1 as hash function for lightweight protocol.

5.3.1 Selection of Pseudo Random Number Generators (PRNGs)

5.3.1.1 Theoretical aspects of PRNGs

A PRNG having the ability to produce genuinely random numbers is a mechanism for generating a sequence of random numbers $U_1, U_2, U_3, \ldots$ with the properties of each U_i distributed between 0 and 1 uniformly and the U_i are independent mutually.

Simple Linear Congruential Generator (LCG) has the mathematical equation as follows:

$$X_{i+1}=(a X_i + c) \bmod m \; ; \quad \text{Here } a - \text{multiplier and } m - \text{modulus}$$

Suitability of a PRNG is evaluated based on its properties such as period length, reproducibility, speed, portability, and randomness.

If c=0 and m is prime, full period is achieved from any $X_0 \neq 0$, if $a^{m-1} - 1$ is a multiple of m and $a^{j-1} - 1$ is not a multiple of m, Then a is called a primitive root of m.

Main consideration in design is to avoid overflow. If ax_i can be represented exactly for every x_i in the sequence, then no overflow occurs. If a is very large then overflow occurs.

We can say the desired properties of a good generator are

- It should efficiently computable, The period should be large, The successive values should be independent and uniformly distributed.
- Selection of LCG parameters
- a,b, and m affect the period and autocorrelation
- The modulus m should be large
- The period can never be more than m.

For 'mod m' computation to be efficient, m should be a power of 2 => 'mod m' can be obtained by truncation.

If b is nonzero, the maximum possible period m is obtained if and only if:

- Integers m and b are relatively prime, that is, have no common factors other than 1
- Every prime number that is a factor of m is also a factor of a-1
- If integer m is a multiple of 4, a-1 should be a multiple of 4.

Notice that all of these conditions are met if $m=2^k$, a=4c+1, and b is odd. Here, c,b, and k are positive integers.

A generator that has the maximum possible period is called a full-period generator.

5.3.1.2 Selection process

The below given table lists the well-accepted PRNGs and values of the constants of well-accepted PRNGs which is used in many popular compilers[53],[38],[45].

	Method	m (modulus)	a (multiplier)	c (increment)
1	LCG Apple Carbon	$2^{31} - 1$	16807	0
2	LCG Borland CPlusPlus	2^{32}	22695477	1
3	LCG CPlusPlusMinstd	$2^{31} - 1$	48271	0
4	LCG FishmanMoore	$2^{31} - 1$	69621	0
5	LCG Formely Common	2^{31}	65539	0
6	LCG GlibC & ANSI C	2^{31}	1103515245	12345
7	LCG ISO IEC	2^{32}	1103515245	12345
8	LCG Marsaglia1	$2^{32}-2$	16807	0
9	LCG Marsaglia2	$2^{32}-5$	69070	0
10	LCG Native API	$2^{31} - 1$	2147483629	2147483587
11	LCG Numerical Recipes	2^{32}	1664525	1013904223
12	LCG Sheffield	2^{31}	16807	0
13	LCG VB6	2^{24}	1140671485	12820163
14	LCG Virtual Pascal	2^{32}	1347758113	1
15	LCG Visual C Plus Plus	2^{32}	214013	2531011
16	LCG VM MTH	2^{32}	69069	1
17	Park Miller	$2^{31} - 1$	16807	0
18	Park Miller Carta	$2^{31} - 1$	16807	0
19	Park Miller New	$2^{31} - 1$	48271	0

Table 5.1 : Values of the constants of well-accepted PRNGs

Jain and Ojha in their two studies[18] and [40] in 2015, identified the Park-Miller PRNG as the best one. They compared Park-Miller's PRNG with other PRNGs such as LCG, Blum Blum Shub, RC4, and Wichman-Hill. Finally, they have concluded that

Park-Miller's PRNGs is the best one for secure lightweight protocols. But, they didn't consider the Park-Miller's latest recommendations in their design.

From above list, PRNG Park-Miller has been chosen as a control PRNG to select an efficient PRNG from the study.

Jain and Ojha have used the Park-Miller's PRNG without considering his latest recommendations. In this study, as per Park-Miller's latest recommendations, an PRNG has been designed as 'Park Miller New'.

Further, PRNGs which have good randomness properties has been chosen for the evaluation purposes. Finally, PRNGs such as LCG GlibC & ANSI C, LCG Marsaglia1, LCG Native API, LCG Sheffield, Park-Miller, Park-Miller-Carta, and Park-Miller New have been chosen for design the secure lightweight protocol.

	Method	m (modulus)	a (multiplier)	c (increment)
1	LCG GlibC & ANSI C	2^{31}	1103515245	12345
2	LCG Marsaglia1	2^{32}-2	16807	0
3	LCG Native API	2^{31} - 1	2147483629	2147483587
4	LCG Sheffield	2^{31}	16807	0
5	Park Miller	2^{31} - 1	16807	0
6	Park Miller Carta	2^{31} - 1	16807	0
7	Park Miller New	2^{31} - 1	48271	0

Table 5.2 : Values of the constants of Selected PRNGs

5.3.2 Selection of Hash functions

Hash functions SHA-1 and SHA 256, SHA384, and SHA512 have been identified to consider in our study. These hash functions have been designed by National Security Agency(NSA) and approved by National Institute of Standards and Technology(NIST), USA and widely used in many applications[49].

Name of Hash Algorithm	Length of Hash Value	
SHA-1	160 bits	Designed by the National Security Agency (NSA) and published by the National Institute of Standards and Technology (NIST) of USA.
SHA 256	256 bits	
SHA 384	384 bits	
SHA 512	512 bits	

Table 5.3 : Properties of the selected Hash Algorithms

5.4 Summary

This chapter discussed the design of the secure lightweight protocol with top level architecture diagram. Top level architecture has three modules such as 'Generate Dynamic Binary Key', 'Encrypt Plain Data' and 'Decrypt Cipher Data'. Architecture diagram of these modules have been presented. Then, theoretical aspects of PRNGs, and basis for the selection of PRNGs and hash functions for this study have been elaborated. Next chapter describes the implementation aspects of the design and implementation for choosing suitable PRNGs and hash functions.

Chapter 6

Implementation of the Efficient and Secure Lightweight Protocol

6.1 Introduction

In chapter 5, the top level design of the efficient and secure lightweight protocol has been described in terms of what each component does. Moreover, the selected PRNGs and hash functions for the evaluation have been described. This chapter describes the implementation of each component regarding software algorithms, platforms etc. In that sense, this chapter is about how the system is implemented.

6.2 Overview of the implementation

Initially, the design described in the previous chapter have been implemented with each of the PRNGs selected with the hash function SHA-1 chosen as control factor. Then the lightweight protocol has been implemented with each of the hash function selected. With seven PRNGs and four hash functions, eleven different lightweight protocols have been implemented.

To collect the execution times of every implementations of the lightweight protocol with different PRNGs and hash functions, Module, 'Automate Data Collection' has been implemented.

MATLAB code of the implementation of all the modules is given in the Appendix–B.

6.3 Software and Platform used for the implementation

The design has been implemented in MATLAB software. The implementation has been evaluated in a PC with MATLAB software. Since the module 'Automate Data Collection' has been needed to run in a less disturbance system for long periods with the facility to confirm generality, It has been implemented in two identical Virtual Machines set up in a Server.

To store the collected data of execution times, MS Excel is used and the module 'Automate Data Collection' has stored all measured times in an Excel sheet for future analysis.

6.4 Implementation of the module 'Generate Dynamic Binary Key'

This module has four main activities such as 'Generate Pseudo Random Number',' 'Generate Matrix'. 'Generate Randomized Matrix', and 'Generate Hash Value'.

'Generate Pseudo Number' for every PRNG has implemented with their relevant constants values and their specific algorithms. 'Generate Matrix' has been developed to combine the two basic keys such as 'User input key' and 'In-built key', and produce a matrix with the size of 14 x 14. By combining the generated matrix and the pseudo random number, 'Generate Randomize Matrix' will produce a randomized matrix with the size of 14 x 14. 'Generate Hash Value' produces a binary stream of 196 bits employing a hash function. This will be used as a Dynamic Binary Key.

Algorithms of the PRNGs such as 'LCG', 'Park-Miller' and 'Park Miller Carta' are given below:

1. Algorithm of LCG :

    function LCGPseudoRandomNumber : real

    // Assume seed is global variable and has an initial value.

    Step 1: Initialize parameters

    a ←48271; m←2147483647; c←4;

    Stept 2: X1← (a x seed + c) mod m

    Step 3: seed ← X1

    Step 4: return ← y1/m

2. Algorithm of ParkMiller :

function ParkMillerPseudoRandomNumber : real

// Assume seed is global variable and has an initial value.

Step 1: Initialize parameters

a ←16807; m←2147483647; q←127773; r←2836;

Step 2: hi ← seed div q

Step 3: lo ← seed mod q

Step 4: test ← a x lo – r x hi

Step 5: if (test>0) then seed ← test

else seed ← test + m

Step 6: return ← seed /m

3. Algorithm of ParkMillerCarta :

function ParkMillerCartaPseudoRandomNumber : real

// Assume seed is global variable and has an initial value.

Step 1: Initialize parameters

a ←16807; m←2147483647;

Step 2: lo ← (a x (seed & 0xFFFF))

Step 3: hi ← (a x (seed >> 16)

Step 4: lo ← lo + (hi & 0x7FFF) <<16

Step 5:lo ← lo + (hi >> 15)

Step 6: if (lo > 0x7FFFFFFF) then lo ← lo - 0x7FFFFFFF

Step 7: seed ← lo

Step 7: return ← lo /m

6.5 Implementation of the function 'EncryptionProcess'

As described in the Chapter 5, the processes in the 'EncryptPlainData', have been implemented in the function name as EncryptionProcess. The function gets plaint data in binary form and encrypt it as cipher data using the key given by the function 'Generate Dynamic Key'.

High level algorithm of this function is given below:

function EncryptionProcess(DynamicBinaryKey,PlainData):CipherData

Step 1. : Get the following sub keys with the length of 49 bits from the different parts of the DynamicBinaryKey.

Sub keys : DP1, DP2, DL1, DL2, DF1, DF2, and DF3

Step 2 : Divide the PlainData as Blocks with the length of 49 bits. If the last the Block doesn't have 49 bits, add pad bits.

Step 3 : For each Blocks perform the following

Step 3.1 : Round 1: Encrypt the Block with the sub keys DP1 and DP2

Between the encryption swap the bits in the Block in a specific manner.

Step 3.2 : Round 2 : Encrypt the Block with the sub key DL1

Step 3.3 : Round 3 : Encrypt the Block with the sub keys DL2 and DF1

Between the encryption process, swap the bits in the Block in a specific manner.

Step 3.4 : Round 4 : Encrypt the Block with the sub keys DF2 and DF3

Between the encryption process, swap the bits in the Block in a specific manner.

Step 4 : Merge the all the Blocks as Cipher Data and return it as output.

6.6 Implementation of the function 'DecryptionProcess'

As described in the Chapter 5, the processes in the 'DecryptCiperData', have been implemented in the function name as DecryptionProcess. The function gets cipher data in binary form and decrypt it as plain data using the key given by the function 'Generate Dynamic Key'.

High level algorithm of this function is given below:

function DecryptionProcess(DynamicBinaryKey,CipherData):PlainData

Step 1. : Get the following sub keys with the length of 49 bits from the different parts of the DynamicBinaryKey.

Sub keys : DKL1, DKF1, DKF2, DKF3, DKP1, and DKP2

Step 2 : Divide the PlainData as Blocks with the length of 49 bits. If the last the Block doesn't have 49 bits, add pad bits.

Step 3 : For each Blocks perform the following

Step 3.1 : Round 1: Decrypt the Block with the sub keys DKF2 and DKF3

Between the encryption swap the bits in the Block in a specific manner.

Step 3.2 : Round 2 : Encrypt the Block with the sub key DL1

Step 3.3 : Round 3 : Encrypt the Block with the sub keys DL2 and DF1

Between the encryption process, swap the bits in the Block in a specific manner.

Step 3.4 : Round 4 : Encrypt the Block with the sub keys DF2 and DF3

Between the encryption process, swap the bits in the Block in a specific manner.

Step 4 : Merge the all the Blocks as Cipher Data and return it as output.

6.7 Implementation of the Module 'AutomateDataCollection_PRNG'

This module has been implemented to automate the operations of collection and storing the measured computation times of the different architectures which have different PRNGs but same hash algorithm. This module has implemented to measure the execution times of the each design for twenty times, calculate the average execution time, and store the values in a MS Excel work sheet.

6.8 Implementation of the Module 'AutomateDataCollection_HF'

This module has been implemented to automate the operations of collection and storing the measured computation times of the different architectures which have different hash algorithms but same PRNG. This module has implemented to measure the execution times of the each design for twenty times, calculate the average execution time, and store the values in a MS Excel work sheet.

6.9 Implementation of the Module 'AutomateFunctionalityTesting'

This module has been implemented to automatically test the functionality of the developed architectures to verify their functionality and give the results in a text file and MS Excel work sheet. Simple and user friendly interface has been created for this module.

6.10 Summary

This chapter presented the implementation aspects of the secure lightweight protocol with different architectures in such a way each one has specific PRNG and hash algorithm. Next chapter describes about the results of the functionality testing of the each implementation and measured execution times of each implementations in numeric and graphical formats.

Chapter 7

Evaluation of the Implemented Protocols

7.1 Introduction

The previous chapter described the implementation of the secure lightweight protocol. This chapter describes how the implementation has been tested and evaluated to identify the efficient PRNG and Hash algorithm. Initially, the implemented eleven models have been tested for their functionality. Later, execution time of the models have been measured and presented in table and graphical formats.

7.2 Testing the functionality of the Implementations

The implementations of the eleven models have been tested to verify its functionality, manually and automated manner. The test results are given in the Appendix - D. Test results ensured the functionality of the implementation are correct.

7.3 Evaluating Strategy

To increase the accuracy in the measurement of execution times, the following matters have been considered in measurements.

- Evaluations have been performed on two identical Virtual Machines having no unnecessary applications or devices. Then, results obtained in two machines have been compared for any considerable differences.
- Minimum size of Input data has been taken as large enough for its execution time has been more than 40 seconds.
- Measuring twenty execution times, average times have been considered for evaluation.
- During the measurement, first measurements have not been considered to find the average to avoid any start up times.
- The built-in function of the MATLAB, 'timeit', which is recommended by MATLAB for robust measurement of the time required for a function execution

and provides a more vigorous estimate, has been utilized for measurement of execution times.

7.4 Average Execution Times of the Implementations with Different PRNGs

To identify the suitable PRNG which provides efficiency to the implementation of the secure lightweight protocol, execution times with different input data have been taken. Seven implementations, each one has different PRNGs but same hash algorithm –SHA-1, have taken for the evaluation. The average execution times calculated are given below with graphical representations. Detail measured times are given in the Appendix - C.

PRNGs	Size 25 Kbyte	Size 30 Kbyte	Size 35 Kbyte	Size 40 Kbyte
LCGGlibC	51.46218	62.00772	73.65602	82.42930
LCGMarsaglia	50.25929	60.32896	70.70038	80.43992
LCGNativeAPI	52.20872	62.94671	73.69287	83.36370
LCGSheffield	49.21276	58.66130	69.15086	78.18048
ParkMiller	52.02353	62.69292	73.33440	82.88175
ParkMillerCarta	51.97637	62.32395	73.09633	83.15086
ParkMillerNew	49.63269	59.47119	70.03893	79.26224

PRNGs	Size 45 Kbyte	Size 50 Kbyte	Size 55 Kbyte
LCGGlibC	93.12619	103.58324	113.92710
LCGMarsaglia	91.03645	101.16679	111.20534
LCGNativeAPI	94.38902	104.63881	115.35699
LCGSheffield	88.45563	98.25484	108.40541
ParkMiller	93.88163	103.86106	114.74446
ParkMillerCarta	93.79393	104.33904	115.07245
ParkMillerNew	91.28653	98.807010	108.96367

Table 7.1 : Measured average execution times for different sizes of input data, with the same hash algorithm.

Graphical representations of the data in the above table are presented in the below given line graphs.

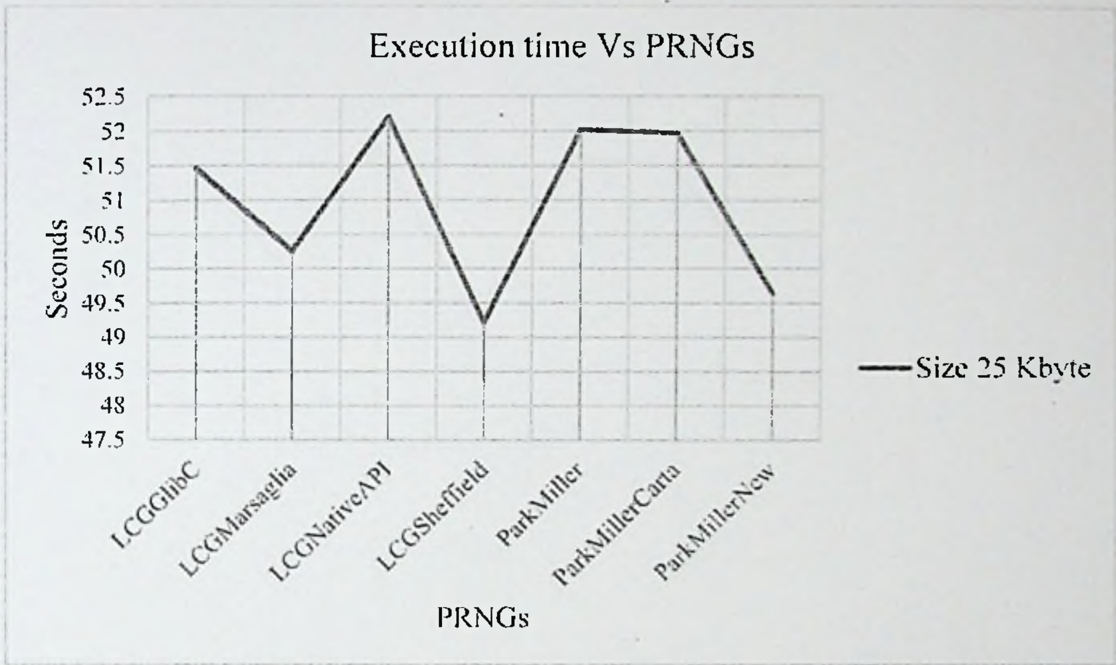


Figure 7.1 : Execution time vs PRNGs for the input data size 25Kbyte

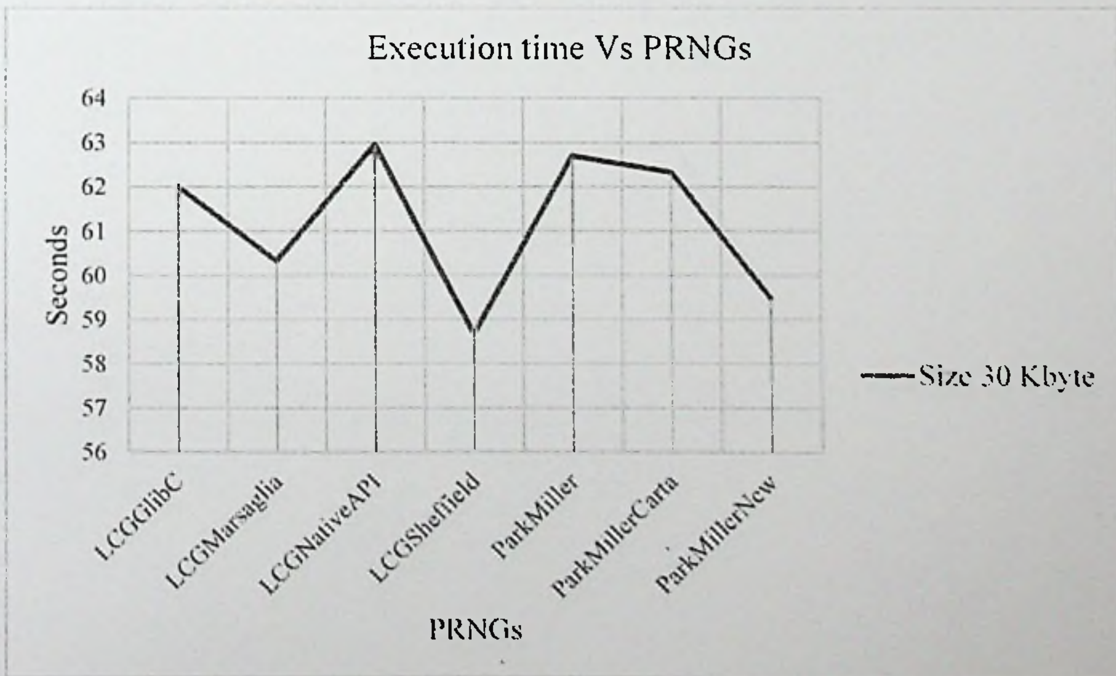


Figure 7.2 : Execution time vs PRNGs for the input data size 30Kbyte

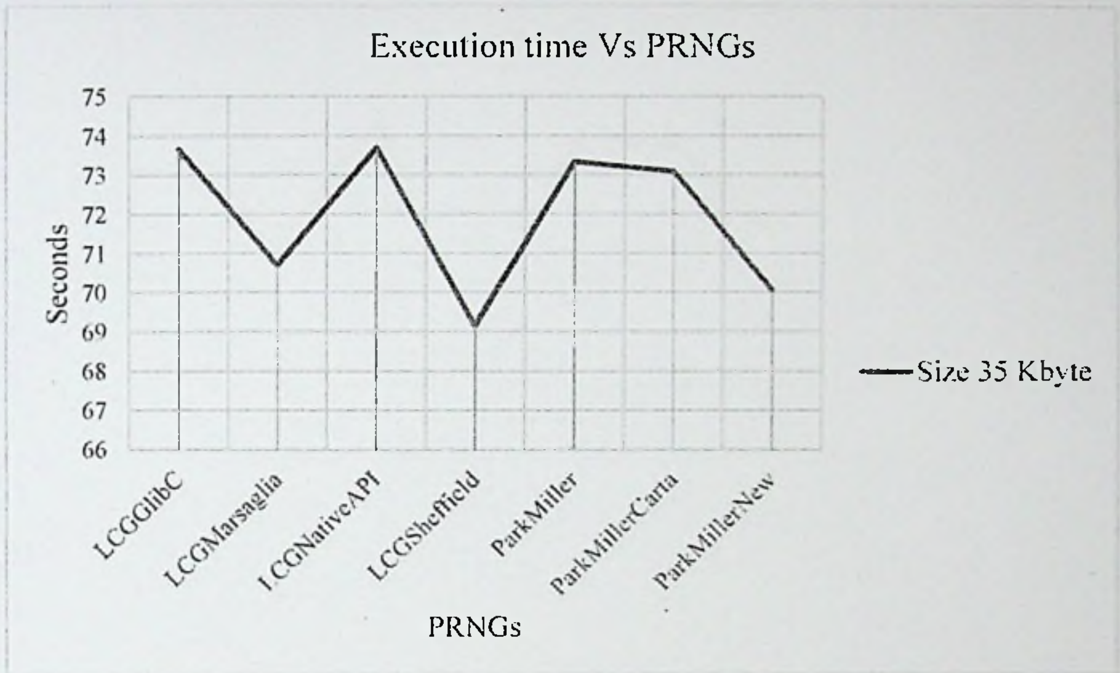


Figure 7.3 : Execution time vs PRNGs for the input data size 35Kbyte

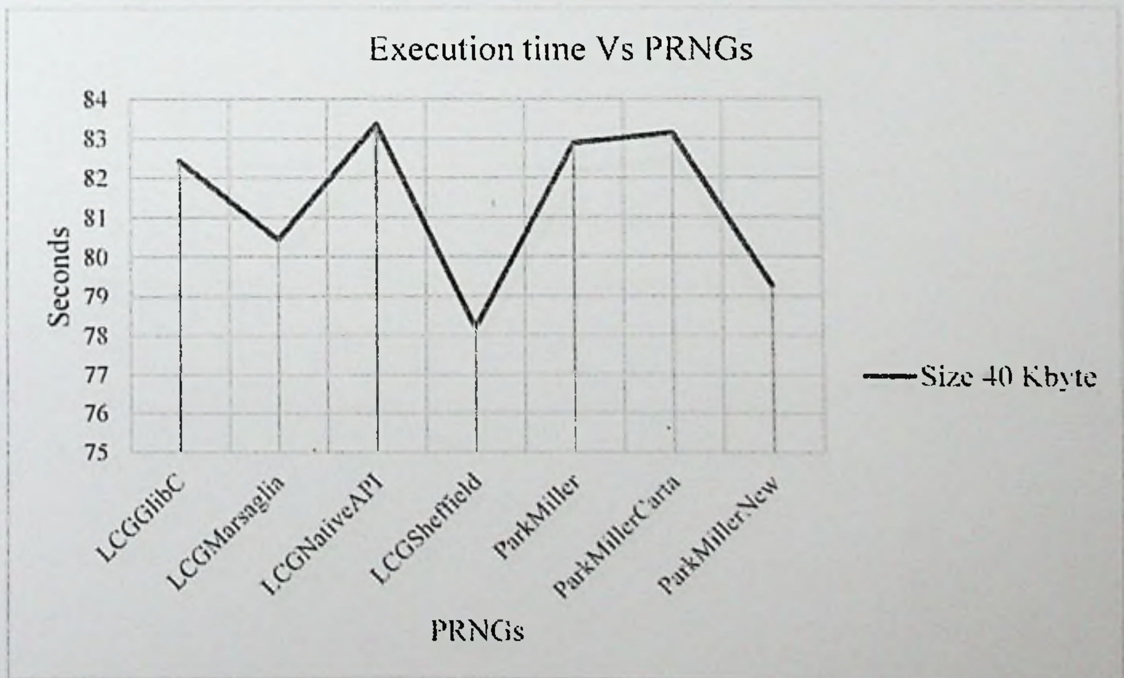


Figure 7.4 : Execution time vs PRNGs for the input data size 40Kbyte

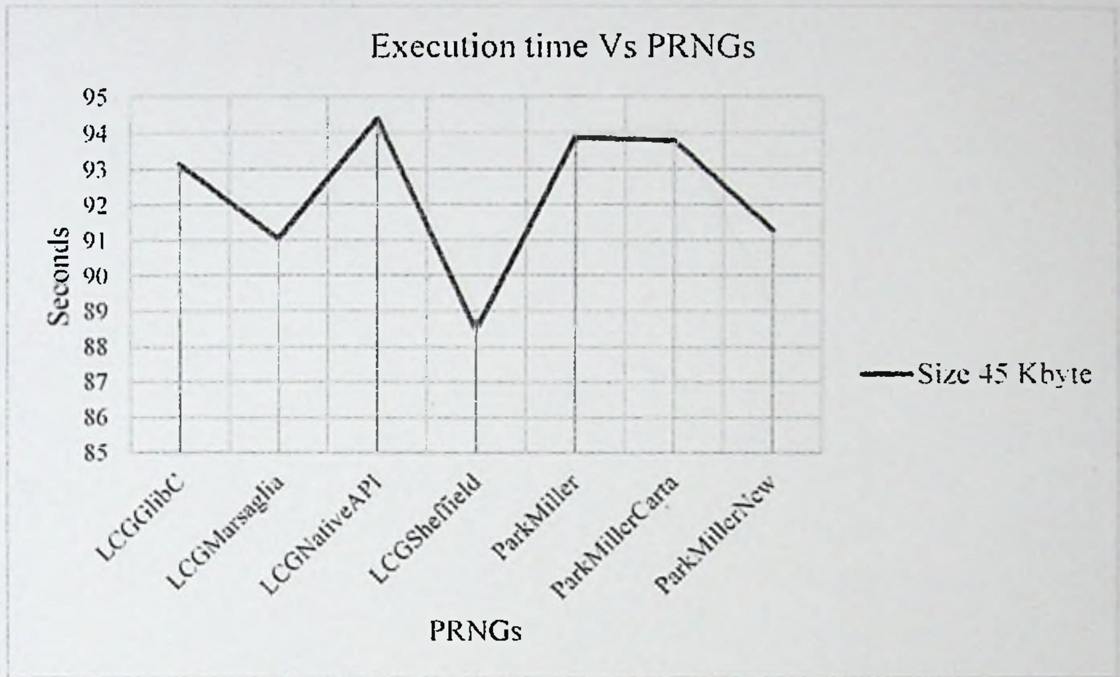


Figure 7.5 : Execution time vs PRNGs for the input data size 45Kbyte

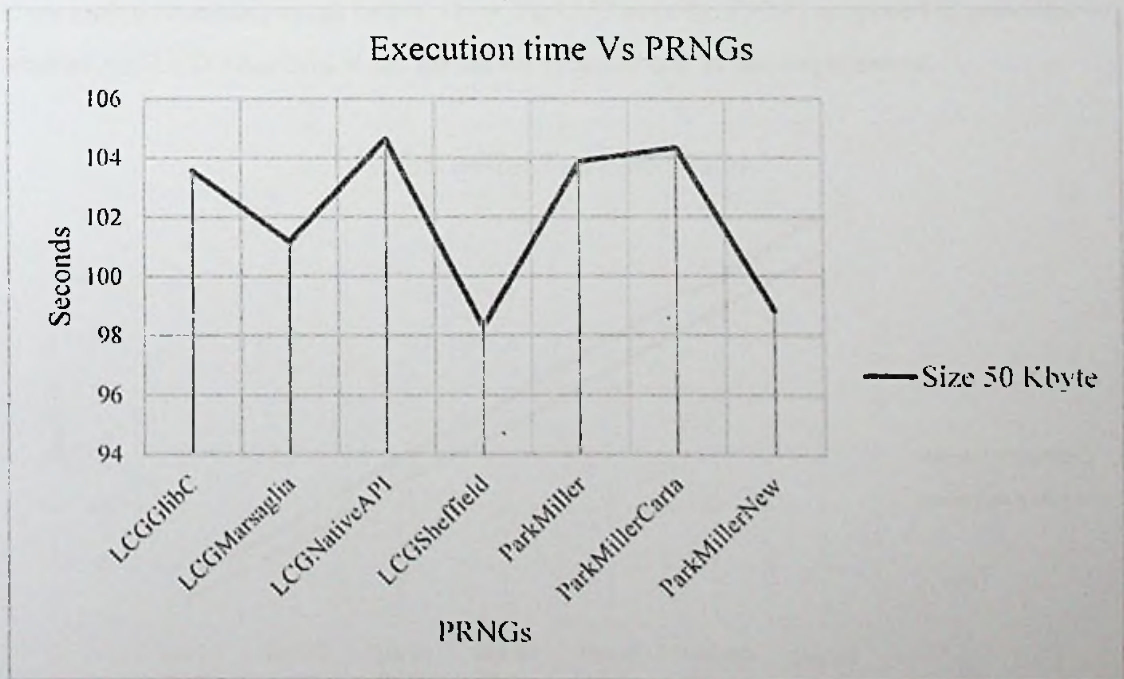


Figure 7.6 : Execution time vs PRNGs for the input data size 50Kbyte

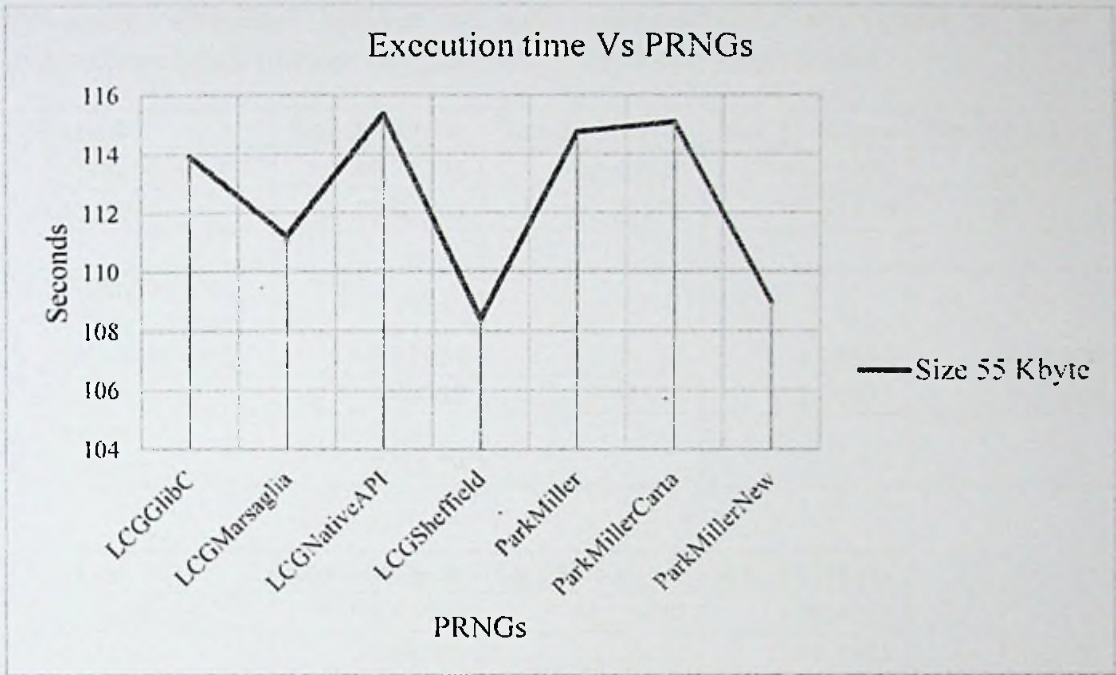


Figure 7.7 : Execution time vs PRNGs for the input data size 55Kbyte

Comparisons between the implementations having ParkMiller and LCG Sheffield in their architectures are given below. Here, ParkMiller is the PRNG proposed in previous studies and LCG Sheffield is the identified efficient one in our experiment.

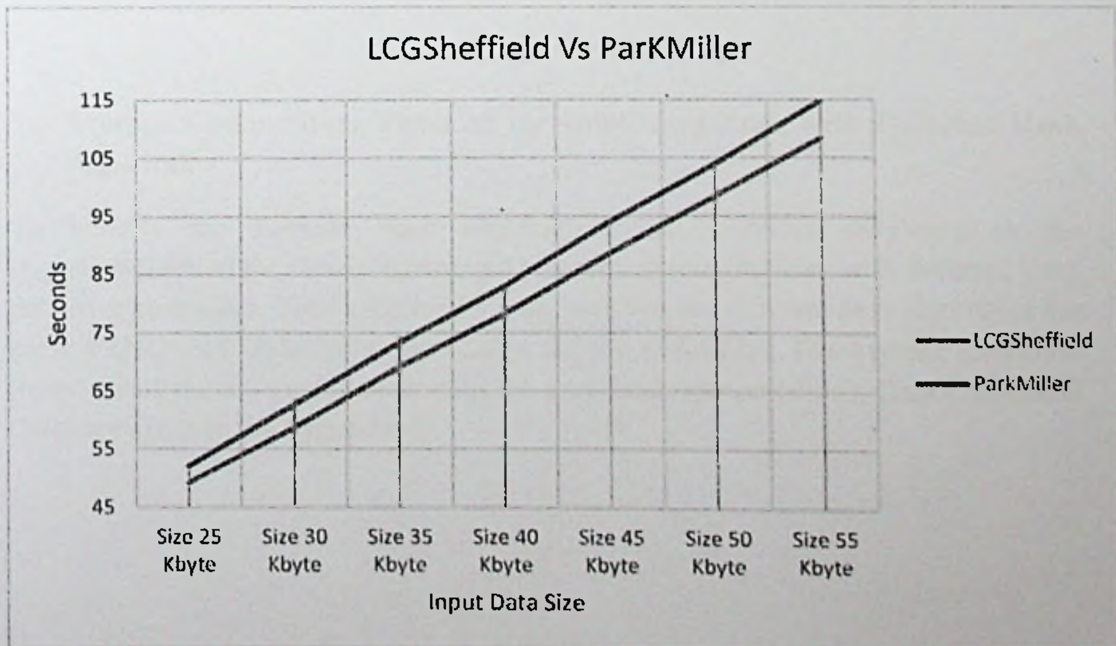


Figure 7.8 : Execution times of LCGSheffield vs ParkMiller PRNGs architectures

Efficiency comparison between the two implementations which have in their architectures ParkMiller and LCGSheffield as PRNG are given below.

Method	Size 25 Kbyte	Size 30 Kbyte	Size 35 Kbyte	Size 40 Kbyte
LCGSheffield	49.21276	58.66129	69.15086	78.18048
ParkMiller	52.02353	62.69292	73.33440	82.88174
LCGSheffield's Efficiency in Sec. against ParkMiller	2.81077	4.03162	4.18354	4.70126
Efficiency in %	5.40288	6.43075	5.70475	5.67226
Average Efficiency	**5.70179**			

Method	Size 45 Kbyte	Size 50 Kbyte	Size 55 Kbyte
LCGSheffield	88.45562	98.25484	108.4054
ParkMiller	93.88163	103.8611	114.7445
LCGSheffield's Efficiency in Sec. against ParkMiller	5.42601	5.60622	6.33905
Efficiency in %	5.77962	5.39781	5.52449

Table 7.2 : Efficiency comparison architectures having LCGSheffield and ParkMiller.

7.5 Average Computation Times of the Implementations with Different Hash Algorithms

To identify the suitable Hash algorithm which provides efficiency to the implementation of the secure lightweight protocol, execution times with different input data have been taken. Four implementations, each one has different hash algorithms but same PRNG - LCGSheffield, have taken for the evaluation. The average execution times calculated are given below with the graphical representations. Detail measured times are given in the Appendix C.

Hash Algorithm	Size 25 Kbyte	Size 30 Kbyte	Size 35 Kbyte	Size 40 Kbyte
SHA-1	48.91310	58.43929	69.48278	78.76926
SHA256	51.03638	61.16325	73.06556	81.79517
SHA384	49.30274	59.19036	70.34567	79.70132
SHA512	49.57987	59.48701	71.21232	80.52878

Hash Algorithm	Size 45 Kbyte	Size 50 Kbyte	Size 55 Kbyte
SHA-1	89.11982	100.55405	110.08024
SHA256	91.24310	104.96086	115.08773
SHA384	89.50947	101.39622	111.28383
SHA512	89.78659	102.11264	112.01979

Table 7.3 : Measured average execution times for different sizes of input data, with the same PRNG.

Graphical representations of the data in the above table are presented in the below given line graphs.

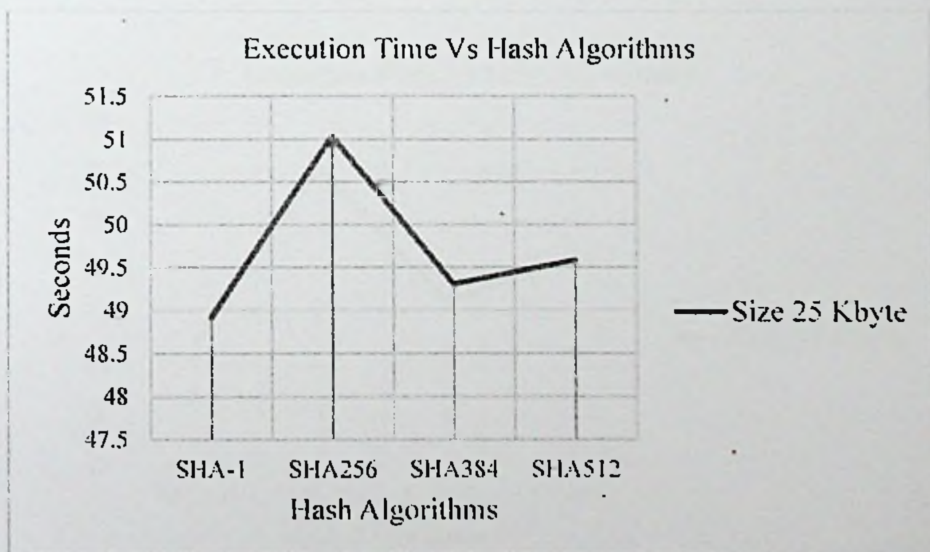


Figure 7.9 : Execution time vs Hash Algorithms for the input data size 25Kbyte

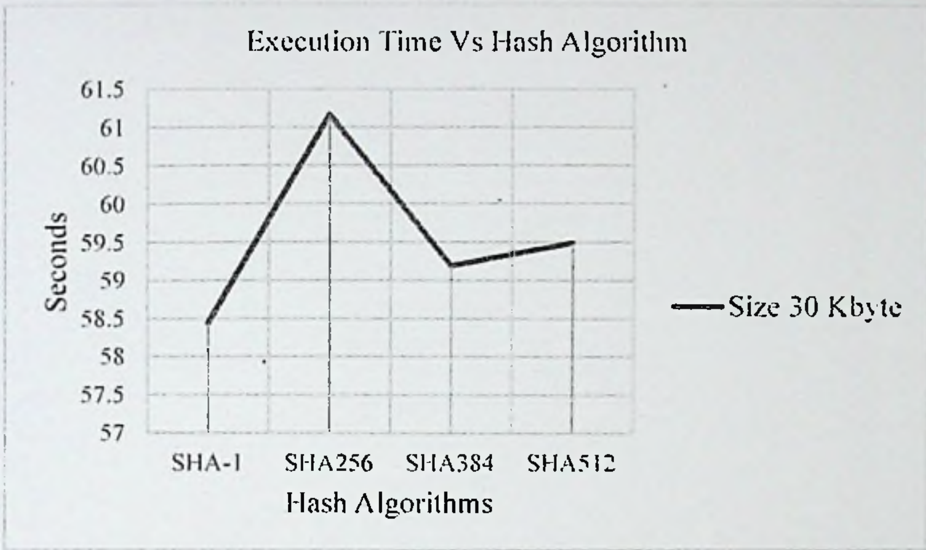


Figure 7.10 : Execution time vs Hash Algorithms for the input data size 30Kbyte

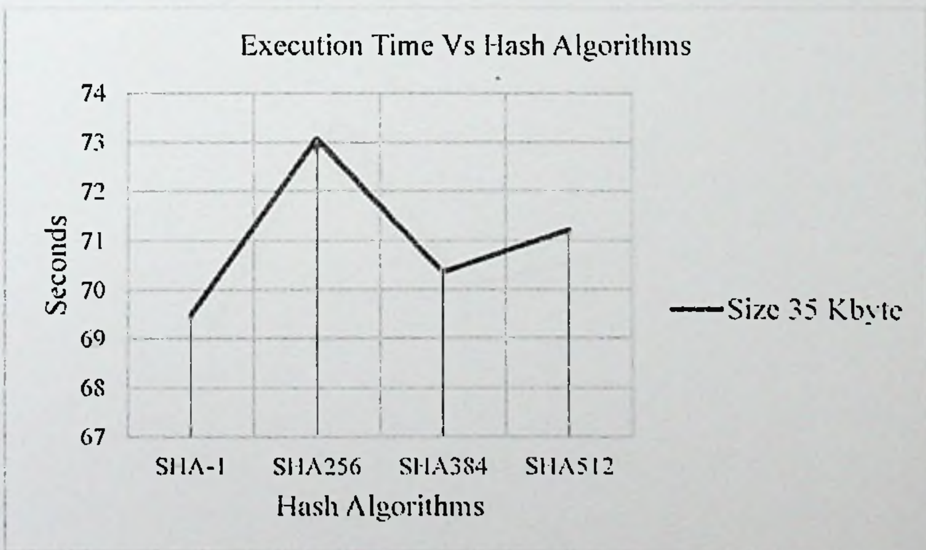


Figure 7.11 : Execution time vs Hash Algorithms for the input data size 35Kbyte

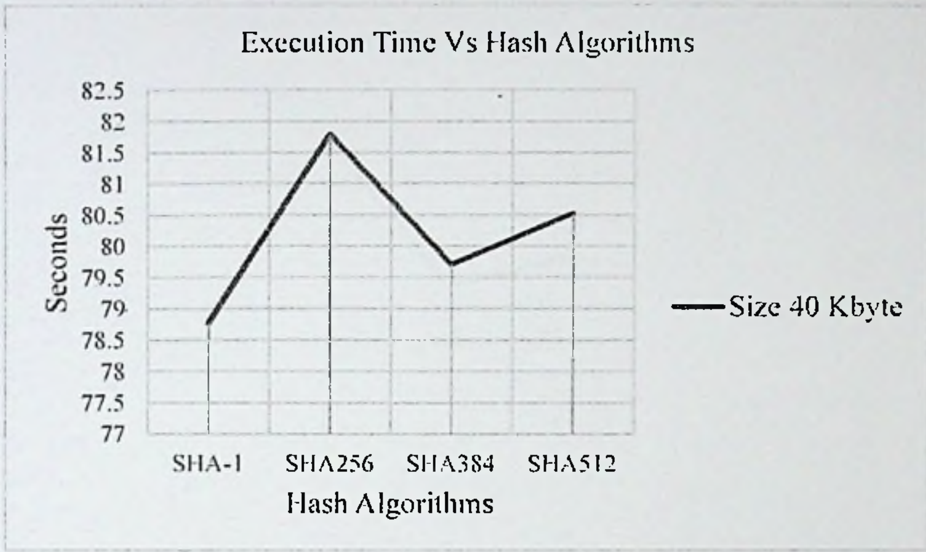


Figure 7.12 : Execution time vs Hash Algorithms for the input data size 40Kbyte

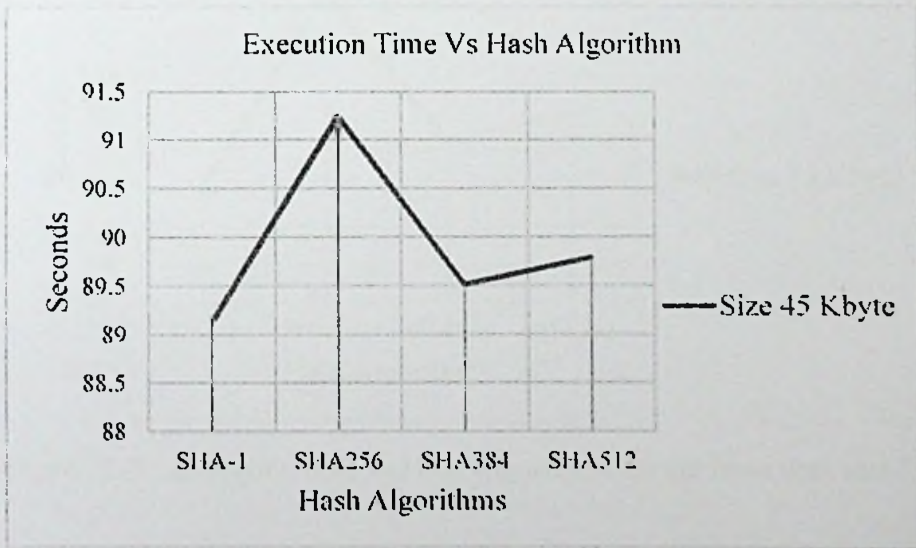


Figure 7.13 : Execution time vs Hash Algorithms for the input data size 45Kbyte

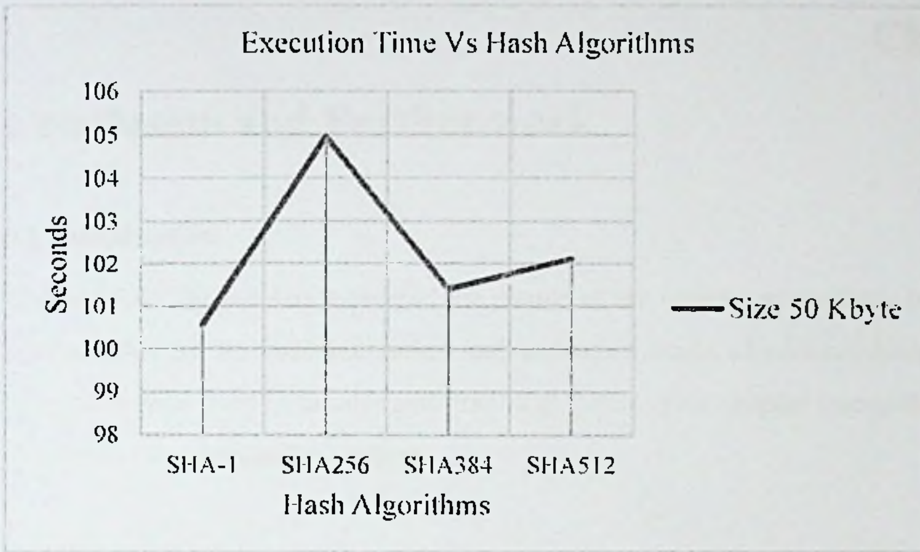


Figure 7.14 : Execution time vs Hash Algorithms for the input data size 50Kbyte

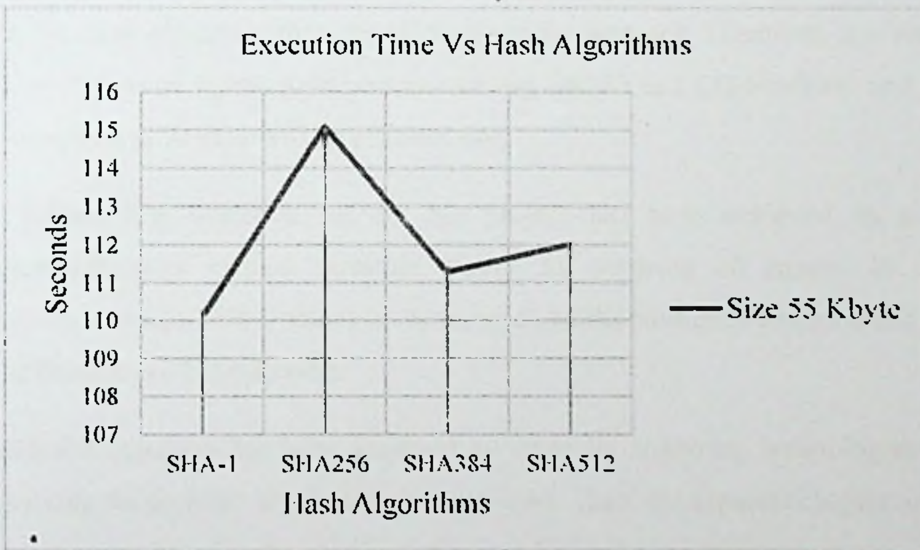


Figure 7.15 : Execution time vs Hash Algorithms for the input data size 55Kbyte

7.6 Summary

This chapter describes the results obtained in the evaluation process. Also, It has expressed the evaluation strategy applied during the evaluation process. The results obtained in the evaluation are shown in tabular form and graphical representation as line graphs. Next chapter will interpret the results described in this chapter in a meaningful way.

Chapter 8

Conclusion and Further work

8.1 Introduction

The previous chapter has presented the results of the various evaluations regarding the functionality of the implementation and execution times of various designs. Results have been expressed in tabular and graphical forms. This chapter interprets the results obtained in the evaluation chapter.

8.2 Conclusion

Based on the analysis, the secure lightweight protocol having a PRNG as LCG Sheffield and secure hash algorithm as SHA-1 is more efficient. This protocol is, in average, 5.7% more efficiency than the existing similar protocol. Therefore, It is recommended that the secure lightweight protocol having PRNG as LCG Sheffield and secure hash algorithm as SHA-1 will be efficient one.

Further, first objective set for this project has been achieved by conducting a comprehensive critical literature survey by covering 20 papers. In this review, achievements and limitations in the research works have been analysed and the problem definition has been derived.

Second objective has been achieved by critically analyzing technologies suitable for solving the problem via the literature reviews. Then, the separate chapter on technology has been written by showing how they can be used to in this project.

Other objectives have been achieved by designing and developing the prototype models and evaluating those models. The prototype development is evidence from three major chapters Approach Design & Implementation. Last objectives have been achieved by evaluating the prototype models and analyzing the results obtained in the evaluation. These have been described in the chapter Evaluation. Also, the research work has been documented in a structured manner.

8.3 Further work

Further study has to be done to propose efficient block size and key size for the secure lightweight protocol for WSNs.

8.4 Summary

This chapter interprets the results obtained in the previous chapter, propose an architecture for efficient and secure lightweight protocol and suggests future studies to improve the secure lightweight protocol.

References

[1] "fred_royan_smart_water_smart_city.pdf." [Online]. Available: http://www.swan-forum.com/uploads/5/7/4/3/5743901/fred_royan_smart_water_smart_city.pdf. [Accessed: 09-Mar-2016].

[2] "Singapore Strives to Become 'The Smartest City.'" [Online]. Available: http://www.governing.com/topics/economic-dev/gov-singapore-smartest-city.html. [Accessed: 09-Mar-2016].

[3] B. C. 11 14 13 7:08 AM, "The 10 Smartest Cities In North America," *Co.Exist*, 14-Nov-2013. [Online]. Available: http://www.fastcoexist.com/3021592/the-10-smartest-cities-in-north-america. [Accessed: 09-Mar -2016].

[4] "Visiting the Smart City of Yinchuan, China," *Amsterdam Smart City*, 08-Oct-2015. [Online]. Available: http://amsterdamsmartcity.com/news/detail/id/711/slug/visiting-the-smart-city-of-yinchuan-china?lang=en. [Accessed: 09-Mar-2016].

[5] "Home," *Smart Cities UK Conference & Expo - Feb 1st & 2nd 2017*. [Online]. Available: http://www.smartcityuk.com/. [Accessed: 09-Mar-2016].

[6] K. Chelli, "Detail Attacks SPINS TINYSEC LEAP descriptions." Proceedings of World Congress on Engineering VOL I, Jul-2015.

[7] A. S. Ahmed, "An Evaluation of Security Protocols on Wireless Sensor Network," in *TKK T-110.5190 Seminar on Internetworking*, 2009.

[8] "Google loon project to cover Sri Lanka with 3G internet." [Online]. Available: http://www.slbc.lk/index.php/tamil-news-update/1361-google-loon-project-to-cover-sri-lanka-with-3g-internet.html. [Accessed: 08-Mar-2016].

[9] "Kandy will be developed as Sri Lanka's first smart city." [Online]. Available: http://www.slbc.lk/index.php/tamil-news-update/1766-kandy-will-be-developed-as-sri-lanka-s-first-smart-city.html. [Accessed: 08-Mar-2016].

[10] "Kandy to be Sri Lanka's first 'smart city' | Colombo Gazette." [Online]. Available: http://colombogazette.com/2015/09/06/kandy-to-be-sri-lankas-first-smart-city/. [Accessed: 08-Mar-2016].

[11] N. O. Nweze, "Real Time Monitoring Of Urban Water Systems for Developing Countries," *IOSR-JCE*, vol. 16, no. 3, May 2014.

[12] L. Cai, R. Wang, J. Ping, Y. Jing, and J. Sun, "Water Supply Network Monitoring Based on Demand Reverse Deduction (DRD) Technology," *Procedia Eng.*, vol. 119, pp. 19–27, 2015.

[13] M. Mutchek and E. Williams, "Moving Towards Sustainable and Resilient Smart Water Grids," *Challenges*, vol. 5, no. 1, pp. 123–137, Mar. 2014.

[14] J. Okache, B. Haggett, and T. Ajmal, "Wireless Sensor Networks for Water Monitoring," *Int. J. Digit. Inf. Wirel. Commun. IJDIWC*, vol. 4315, p. 349360, 2012.

[15] T. R. Patil and R. M. Khaire, "Wireless Sensor Network for Real Time Monitoring and Detection of Water Contamination," *IJSRM*, vol. 3, no. 6, Jun. 2015.

[16] B. Sun, C.-C. Li, K. Wu, and Y. Xiao, "A lightweight secure protocol for wireless sensor networks," *Comput. Commun.*, vol. 29, no. 13, pp. 2556–2568, 2006.

[17] N. Bharatesh and S. Rohith, "FPGA Implementation of Park-Miller Algorithm to Generate Sequence of 32-Bit Pseudo Random Key for Encryption and Decryption of Plain Text," *Int. J. Reconfigurable Embed. Syst. IJRES*, vol. 2, no. 3, pp. 99–105, 2013.

[18] K. Jain and A. Ojha, "Implementation of LiSP using Park-Miller for Wireless Sensor Network," *Int. J. Comput. Appl.*, vol. 110, no. 8, 2015.

[19] A. Perrig, J. Stankovic, and D. Wagner, "Security in wireless sensor networks," *Commun. ACM*, vol. 47, no. 6, pp. 53–57, 2004.

[20] M. Brown, D. Cheung, D. Hankerson, J. L. Hernandez, M. Kirkup, and A. Menezes, "PGP in Constrained Wireless Devices.," in *USENIX Security Symposium*, 2000.

[21] S. Chakraborty and V. Kumar, "A Study and Implementation of RSA Cryptosystem," *ArXiv Prepr. ArXiv150604265*, 2015.

[22] D. W. Carman, P. S. Kruus, and B. J. Matt, "Constraints and approaches for distributed sensor network security (final)," *DARPA Proj. ReportCryptographic Technol. Group Trust. Inf. Syst. NAI Labs*, vol. 1, no. 1, 2000.

[23] P. Ganesan, R. Venugopalan, P. Peddabachagari, A. Dean, F. Mueller, and M. Sichitiu, "Analyzing and modeling encryption overhead for sensor network nodes," in *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, 2003, pp. 151–159.

[24] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. CRC press, 1996.

[25] C. Karlof, N. Sastry, and D. Wagner, "TinySec: a link layer security architecture for wireless sensor networks," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*, 2004, pp. 162–175.

[26] L. E. Lightfoot, J. Ren, and T. Li, "An Energy efficient link-layer security protocol for wireless sensor networks." May-2007.

[27] A. Perrig, R. Szewczyk, J. D. Tygar, V. Wen, and D. E. Culler, "SPINS: Security protocols for sensor networks," *Wirel. Netw.*, vol. 8, no. 5, pp. 521–534, 2002.

[28] T. Park and K. G. Shin, "LiSP: A lightweight security protocol for wireless sensor networks," *ACM Trans. Embed. Comput. Syst. TECS*, vol. 3, no. 3, pp. 634–660, 2004.

[29] K. Ren, W. Lou, and Y. Zhang, "LEDS: Providing Location-aware End-to-end Data Security in Wireless Sensor Networks." IEEE, Apr-2006.

[30] J. Ren, T. Li, and D. Aslam, "A Power Efficient Link-Layer Security Protocol (LLSP) for Wireless Sensor Networks." IEEE, Oct-2005.

[31] J. Undercoffer, S. Avancha, A. Joshi, and J. Pinkston, "Security for sensor networks," in *CADIP Research Symposium*, 2002, pp. 25–26.

[32] R. A. Shaikh, S. Lee, M. A. Khan, and Y. J. Song, "LSec: lightweight security protocol for distributed wireless sensor network," in *Personal wireless communications*, 2006, pp. 367–377.

[33] Z. Mahmood, A. Jain, and C. Agrawal, "Hybridize Dynamic Symmetric Key Cryptography using LCG," *Int. J. Comput. Appl.*, vol. 60, no. 17, 2012.

[34] R. Oppliger, *Contemporary cryptography*. Boston: Artech House, 2005.

[35] J. Daemen and V. Rijmen, "AES proposal: Rijndael," 1999.

[36] B. Schneier, "Description of a new variable-length key, 64-bit block cipher (Blowfish)," in *Fast Software Encryption*, 1993, pp. 191–204.

[37] Z. Mahmood, J. L. Rana, and A. Khare, "Symmetric Key Cryptography using Dynamic Key and Linear Congruential Generator (LCG)," *Int. J. Comput. Appl.*, vol. 50, no. 19, 2012.

[38] S. K. Park and K. W. Miller, "Random number generators: good ones are hard to find," *Commun. ACM*, vol. 31, no. 10, pp. 1192–1201, 1988.

[39] C.-C. Lin, S. Shieh, and J.-C. Lin, "Lightweight, Distributed Key Agreement Protocol for Wireless Sensor Networks," 2008, pp. 96–102.

[40] A. Ojha and K. Jain, "Implementation of LiSP using Different Random Number Generator as a Dynamic Key for Wireless Sensor Network," *IJARCCE*, pp. 420–425, Feb. 2015.

[41] R. Basu, S. Ganguly, S. Maitra, and G. Paul, "A complete characterization of the evolution of RC4 pseudo random generation algorithm." Jan-2008.

[42] L. Blum, M. Blum, and M. Shub, "A simple unpredictable pseudo-random number generator," *SIAM J. Comput.*, vol. 15, no. 2, pp. 364–383, 1986.

[43] W. N. Graham, "A comparison of four pseudo random number generators implemented in Ada," *ACM SIGSIM Simul. Dig.*, vol. 22, no. 2, pp. 3–18, 1992.

[44] H. Bauke, *Tina's random number generator library*. August, 2011.

[45] D. F. Carta, "Two fast implementations of the 'minimal standard' random number generator," *Commun. ACM*, vol. 33, no. 1, pp. 87–88, 1990.

[46] "Chapter_16 - 1710027857Chapter_16.pdf." [Online]. Available: http://www.ijsr.in/upload/1710027857Chapter_16.pdf. [Accessed: 24-Mar-2016].

[47] K. Jain and A. Ojha, "A Survey on Lightweight Protocol Using Dynamic Key for Wireless Sensor Network," *IJARCCE*, vol. 3, no. 9, Sep. 2014.

[48] S. U. Rehman, M. Bilal, B. Ahmad, K. M. Yahya, A. Ullah, and O. U. Rehman, "Comparison based analysis of different cryptographic and encryption techniques using message authentication code (mac) in wireless sensor networks (wsn)," *ArXiv Prepr. ArXiv12033103*, 2012.

[49] "NIST.gov - Computer Security Division - Computer Security Resource Center." [Online]. Available: http://csrc.nist.gov/groups/ST/hash/policy.html. [Accessed: 07-Mar-2016].

[50] S. Sridharan, "Water Quality Monitoring System Using Wireless Sensor Network," *Int. J. Electron. Commun. Eng. Adv. Res.*, vol. 3, pp. 399–402, 2014.

[51] G. V. and K. Chandrasekaran, "A Distributed Trust Based Secure Communication Framework for Wireless Sensor Network," *Wirel. Sens. Netw.*, vol. 06, no. 09, pp. 173–183, 2014.

[52] J. Singh, K. Lata, and J. Ashraf, "Image Encryption & Decryption with Symmetric Key Cryptography using MATLAB," *Int. J. Curr. Eng. Technol.*, vol. 5, no. 1, pp. 448–451, 2015.

[53] "Linear congruential generator explained." [Online]. Available: http://everything.explained.today/Linear_congruential_generator/. [Accessed: 05-Mar-2016].

Appendix A – Detail design diagram

The detailed diagrams of the design of the architecture are given below. It is the continuation of the diagrams given in the Chapter 5.

The below given diagrams of sub modules are of the module 'Encrypt Plain Data' described in the chapter 5.

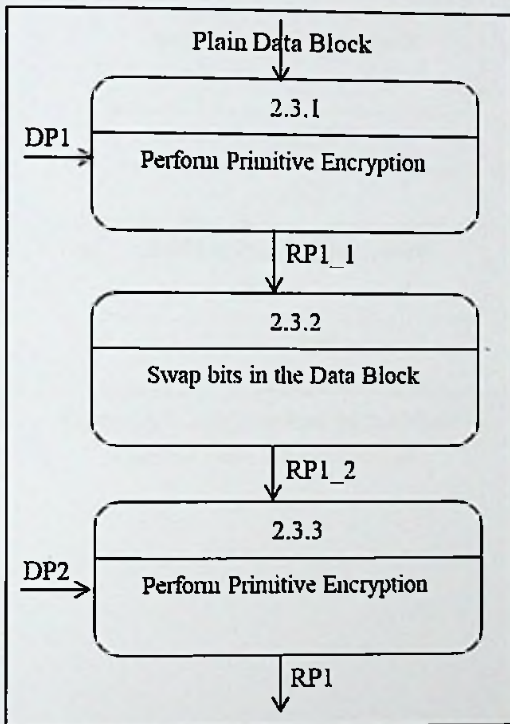


Figure A.1 : Architecture of the Module 'Encrypt Data Block : Round 1'

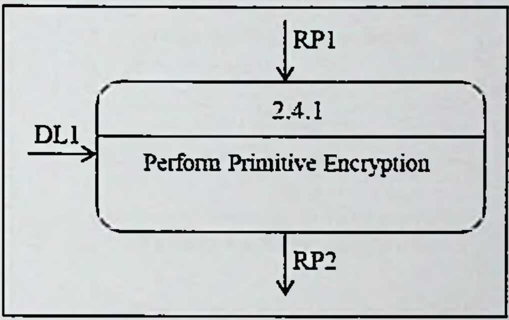


Figure A.2 : Architecture of the Module 'Encrypt Data Block : Round 2'

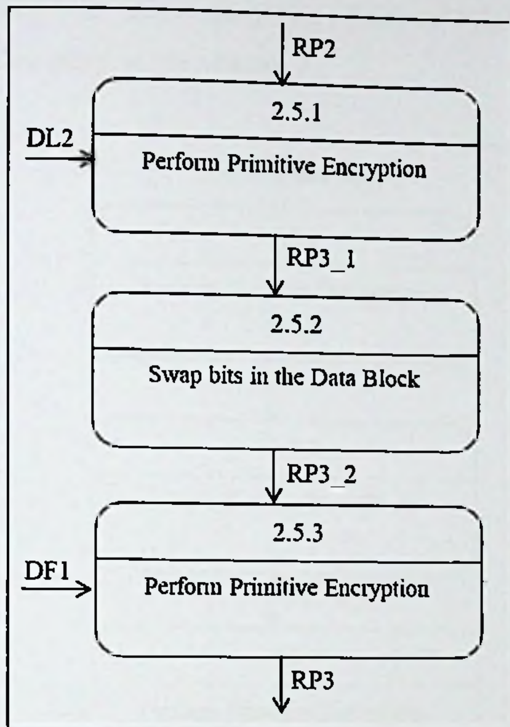


Figure A.3 : Architecture of the Module 'Encrypt Data Block : Round 3'

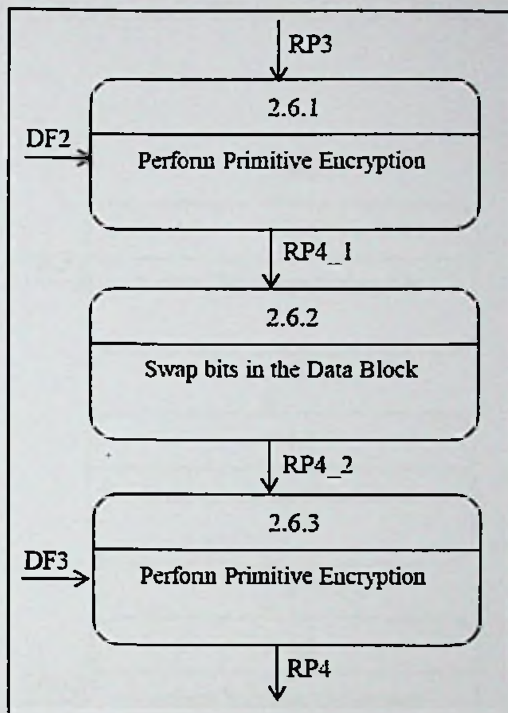


Figure A.4 : Architecture of the Module 'Encrypt Data Block : Round 4'

The below given diagrams of sub modules are of the module 'Decrypt Cipher Data' described in the chapter 5.

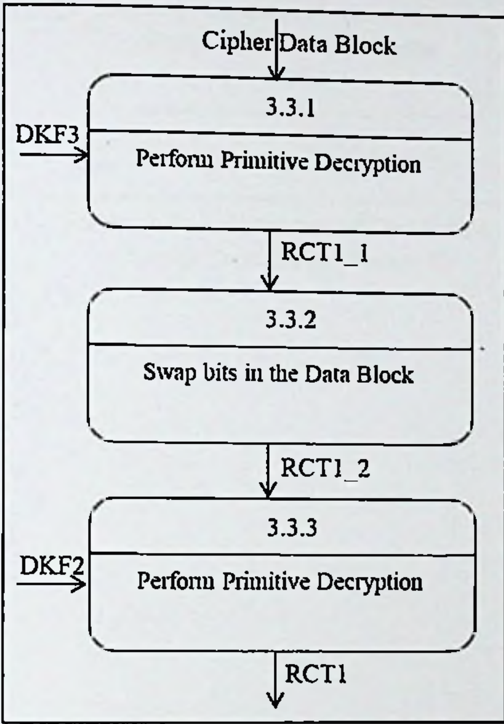


Figure A.5 : Architecture of the Module 'Decrypt Data Block : Round 1'

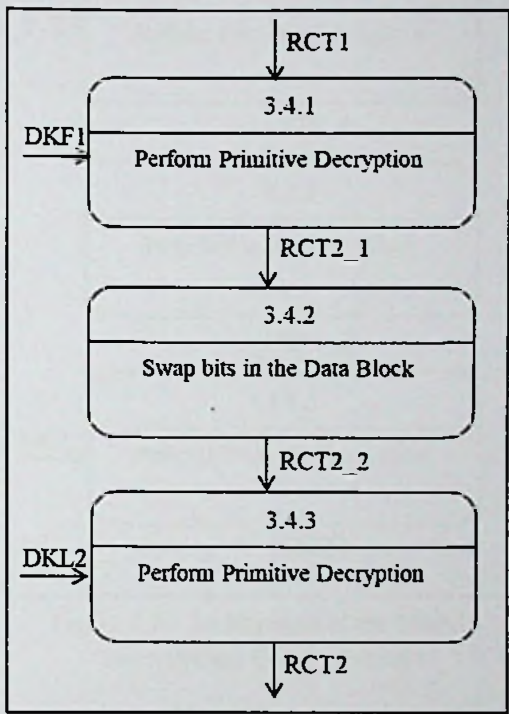


Figure A.6 : Architecture of the Module 'Decrypt Data Block : Round 2'

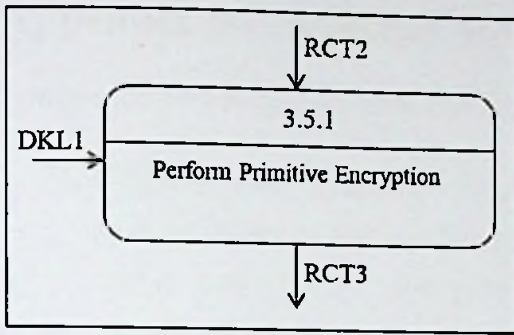


Figure A.7 : Architecture of the Module 'Decrypt Data Block : Round 3'

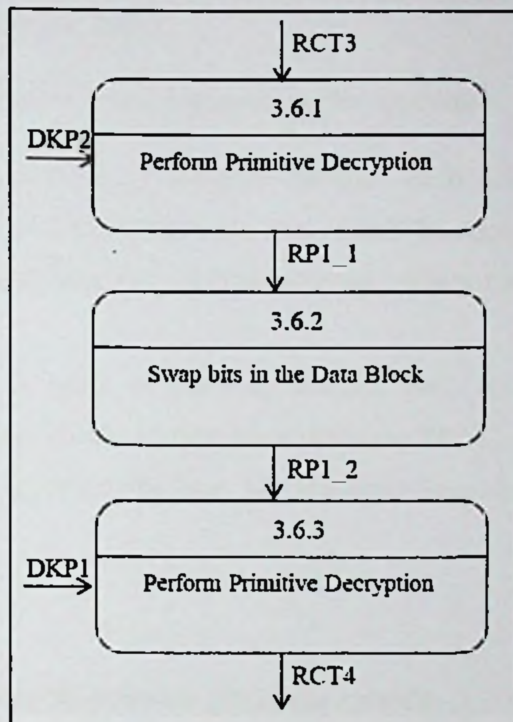


Figure A.8 : Architecture of the Module 'Decrypt Data Block : Round 4'

Appendix B – Selected Source Code

Source code written in MATLAB for the following tasks are given in this appendix.

1. Measuring and storing the execution times of different designs, each one utilizing different PRNGs. But all the designs have the same Hash function, SHA-1. :- B.1 Listing 'AutomateDataCollection_PRNGEvaluation' (Only the specific modules)
2. Measuring and storing the execution times of different designs, each one utilizing different Hash functions. But all the designs have the same PRNG - LCGSheffield. :– B.2 Listing 'AutomateDataCollection_HF' (Only the Specific modules)

B.1 Listing 'AutomateDataCollection_PRNGEvaluation' (Only the specific modules)

```
function AutomateDataCollection_PRNGEvaluation

% This module collects the execution times of the selected designs of the

% protocol with different sizes of input data. For this purpose, it calls

% the module -'MeasuringAllDesigns' - with the different multiples of the

% basic plain data.


%Size of the Basic Plain Data is 160 bytes

BasicPlainData1='WSNs are primarily designed for monitoring environments that humans cannot easily reach (e.g., motion, target tracking, fire detection, chemicals, temperature).';

BasicPlainData='cbs sfdgsd 3246 fdsd vcbx ';


MeasuringAllDesigns_PRNG(2,BasicPlainData,'S160');


% MeasuringAllDesigns_PRNG(160,BasicPlainData,'S160');

% MeasuringAllDesigns_PRNG(192,BasicPlainData,'S192');

% MeasuringAllDesigns_PRNG(224,BasicPlainData,'S224');
```

```
% MeasuringAllDesigns_PRNG(256,BasicPlainData,'S256');
% MeasuringAllDesigns_PRNG(288,BasicPlainData,'S288');
% MeasuringAllDesigns_PRNG(320,BasicPlainData,'S320');
% MeasuringAllDesigns_PRNG(352,BasicPlainData,'S352');
%
disp('Automated Data Collection has been completed....!');


function  MeasuringAllDesigns_PRNG(SN,Str,FN)
% SN - Number of multiples of the Basic Plain Data which is required for
% encryption process; Str - Basic Plain Data; FN - String assigned to the
% end of file name which will have the collected data


% This module evaluates all the selected designs with different PRNGs but
% with same Hash function SHA-1


global seed; % Initial value for Pseudo Random Number Generator
global MesBlockSize; % Size of block in bits
global KeyLength; % Length of key in bits


seed=17; % Assigning initial value for seed


MesBlockSize=49;
KeyLength=196;


InputKey='A&8RT98'; % User input key
%Fixed built-in key for testing purposes; 16 real numbers
IBK=[0.9499    0.4010    0.9200    0.3802    0.9078    0.8893    0.8221    0.7610
0.6723    0.3033    0.9393 0.9699    0.5696    0.6978    0.3249    0.2205];


% Preparing message as per required size that is multiples of SN, Basic
```

```
% Plain Data
PlainText='';
for i=1:SN
   PlainText=strcat(PlainText, Str);
end;

PlainData=Text2Binary(PlainText); % Converting the Plain Data which is in
% ASCII form to an binary form, Since the design will support only binary
% stream

RequiredIterations=1; % By performing more iterations, accuracy can be
                 % improved
%To avoid initial start up distrubances in the time for considerations
%Additional iterations are used
AdditionalIterations=1;
N=RequiredIterations+AdditionalIterations;

% Number of Designs is planned to test in this automation
NumberOfDesigns=7;

%Number of Rows will give space between collected data and the calculated
%average values in the MS Excel Sheet.
NumberOfBlankRows=5;

% Required Matrix size to store the measured execution times
RowsInMatrix=N+NumberOfBlankRows;

% Creating matrix to store the results
MeasuredTimes=zeros(RowsInMatrix,NumberOfDesigns);
```

```
% Keeping the initial seed in a temporary location to ensure each designs
% to use the same seed during the execution
TempSeed=seed;

for k=1:N
   x1=0;
   x2=0;
   x3=0;
   x4=0;
   x5=0;
   x6=0;
   x7=0;

   seed=TempSeed;
   f = @() Design_PRNGGlibC(IBK,InputKey,PlainData); % handle to function
   x1=x1+timeit(f);

   seed=TempSeed;
   f = @() Design_PRNGMarsaglia(IBK,InputKey,PlainData); % handle to function
   x2=x2+timeit(f);

   seed=TempSeed;
   f = @() Design_PRNGNativeAPI(IBK,InputKey,PlainData); % handle to function
   x3=x3+timeit(f);

   seed=TempSeed;
   f = @() Design_PRNGSheffield(IBK,InputKey,PlainData); % handle to function
   x4=x4+timeit(f);
```

```
seed=TempSeed;
f = @() Design_PRNGParkMiller(IBK,InputKey,PlainData); % handle to function
x5=x5+timeit(f);


seed=TempSeed;
f = @() Design_PRNGParKMillerCarta(IBK,InputKey,PlainData); % handle to
function
x6=x6+timeit(f);


seed=TempSeed;
f = @() Design_PRNGParkMillerNew(IBK,InputKey,PlainData); % handle to
function
x7=x7+timeit(f);


j=k+NumberOfBlankRows;
% Storing the measured execution times in the Matrix
MeasuredTimes(j,1)=x1;
MeasuredTimes(j,2)=x2;
MeasuredTimes(j,3)=x3;
MeasuredTimes(j,4)=x4;
MeasuredTimes(j,5)=x5;
MeasuredTimes(j,6)=x6;
MeasuredTimes(j,7)=x7;


count=k %Just to display the current iteration number during the
        %testing process to monitor the progress
TempSeed=seed; % Changing the initial seed for next round..THis
           % ensures in every rounds all designs will get the same
           % seed
```

```
end;

% max and min are used to keep the maximum and minimum average execution
% times
max=-99;
min=99;
for k=1:NumberOfDesigns
   MeasuredTimes(1,k)=0;
   for i=(NumberOfBlankRows+AdditionalIterations+1):RowsInMatrix
      MeasuredTimes(1,k)=MeasuredTimes(1,k)+MeasuredTimes(i,k);
   end;
   MeasuredTimes(1,k)=MeasuredTimes(1,k)/RequiredIterations;
   if (MeasuredTimes(1,k)>max)
      max=MeasuredTimes(1,k);
   end;
   if (MeasuredTimes(1,k)<min)
      min=MeasuredTimes(1,k);
   end;
   scount=k
end;
MeasuredTimes(2,1)=max;
MeasuredTimes(2,2)=min;
MeasuredTimes(2,3)=max-min;

% The measure values in the Matrix is written into a MS Excel Sheet with
% the required file name

FN=strcat('TimeAnalysisDifferent_PRNG',FN);
```

```
FN=strcat(FN,'.xlsx');
filename = FN;
xlswrite(filename,MeasuredTimes);

function Design_PRNGGlibC(IBK,InputKey,PlainData)
% This Module have the design using PRNG as LCGGlibC and Hash function
% as SHA-1

% Global variable 'seed' must have initial value to generate X1
X1=LCGRandomNumberGlibC;
Key=GenerateBinaryKey(X1, IBK, InputKey);

ChiperData=EncryptionProcess(Key,PlainData); % Encrypting PlainData
PlainData=DecryptionProcess(Key,ChiperData); % Decrypting ChiperData
%Message6=Binary2Text(PlainData) %This is for verifying the functionality

function Design_PRNGMarsaglia(IBK,InputKey,PlainData)
% This Module have the design using PRNG as LCGMarsaglia and Hash function
% as SHA-1

% Global variable 'seed' must have initial value to generate X1
X1=LCGRandomNumberMarsaglia;
Key=GenerateBinaryKey(X1, IBK, InputKey);

ChiperData=EncryptionProcess(Key,PlainData); % Encrypting PlainData
PlainData=DecryptionProcess(Key,ChiperData); % Decrypting ChiperData
%Message8=Binary2Text(PlainData) %This is for verifying the functionality

function Design_PRNGNativeAPI(IBK,InputKey,PlainData)
```

```
% This Module have the design using PRNG as LCGNativeAPI and Hash function
% as SHA-1

% Global variable 'seed' must have initial value to generate X1
X1=LCGRandomNumberNativeAPI;
Key=GenerateBinaryKey(X1, IBK, InputKey);

ChiperData=EncryptionProcess(Key,PlainData); % Encrypting PlainData
PlainData=DecryptionProcess(Key,ChiperData); % Decrypting ChiperData
%Message10=Binary2Text(PlainData) %This is for verifying the functionality

function Design_PRNGParkMiller(IBK,InputKey,PlainData)
% This Module have the design using PRNG as ParkMiller and Hash function
% as SHA-1

% Global variable 'seed' must have initial value to generate X1
X1=ParkMillerRandomNumber;
Key=GenerateBinaryKey(X1, IBK, InputKey);

ChiperData=EncryptionProcess(Key,PlainData); % Encrypting PlainData
PlainData=DecryptionProcess(Key,ChiperData); % Decrypting ChiperData
%Message17=Binary2Text(PlainData) %This is for verifying the functionality

function Design_PRNGParKMillerCarta(IBK,InputKey,PlainData)
% This Module have the design using PRNG as ParkMillerCarta and Hash function
% as SHA-1

% Global variable 'seed' must have initial value to generate X1
X1=ParkMillerRandomNumberCarta;
```

```
Key=GenerateBinaryKey(X1, IBK, InputKey);

ChiperData=EncryptionProcess(Key,PlainData); % Encrypting PlainData
PlainData=DecryptionProcess(Key,ChiperData); % Decrypting ChiperData
%Message18=Binary2Text(PlainData) %This is for verifying the functionality

function Design_PRNGParkMillerNew(IBK,InputKey,PlainData)
% This Module have the design using PRNG as ParkMillerNew and Hash function
% as SHA-1

% Global variable 'seed' must have initial value to generate X1
X1=ParkMillerRandomNumberNew;
Key=GenerateBinaryKey(X1, IBK, InputKey);

ChiperData=EncryptionProcess(Key,PlainData); % Encrypting PlainData
PlainData=DecryptionProcess(Key,ChiperData); % Decrypting ChiperData
%Message20=Binary2Text(PlainData) %This is for verifying the functionality

function Design_PRNGSheffield(IBK,InputKey,PlainData)
% Global variable 'seed' must have initial value to generate X1
X1=LCGRandomNumberSheffield;
Key=GenerateBinaryKey(X1, IBK, InputKey);

ChiperData=EncryptionProcess(Key,PlainData); % Encrypting PlainData
PlainData=DecryptionProcess(Key,ChiperData); % Decrypting ChiperData
%Message12=Binary2Text(PlainData) %This is for verifying the functionality

function RN=LCGRandomNumberGlibC
% Rand number generation based on LCG Method(32 bit precision)
```

```
% Before calling this function, seed has to be initialized
% LCGRandomNumberGlibC  & ANSI C

global seed;

a= 1103515245;
m= 2147483648;
c=12345;

y0=seed;
y1=mod(((a*y0)+c), m);
seed=y1;

RN= y1/m;

function RN=LCGRandomNumberMarsaglia
% Rand number generation based on LCG Method(32 bit precision)
% Before calling this function, seed has to be initialized
% LCGRandomNumber Marsaglia1

global seed;

a= 16807;
m= 4294967294;
c=0;

y0=seed;
y1=mod((a*y0), m);
seed=y1;
```

```
RN= y1/m;

function RN=LCGRandomNumberNativeAPI
% Rand number generation based on LCG Method(32 bit precision)
% Before calling this function, seed has to be initialized
% LCGRandomNumberNativeAPI

global seed;

a= 2147483629;
m= 2147483647;
c=2147483587;

y0=seed;
y1=mod((a*y0+c), m);
seed=y1;

RN= y1/m;

function RN=LCGRandomNumberSheffield
% Rand number generation based on LCG Method(32 bit precision)
% Before calling this function, seed has to be initialized
% LCGRandomNumber Sheffield

global seed;

a= 16807;
m= 2147483648;
```

```
c=0;

y0=seed;
y1=mod((a*y0), m);
seed=y1;

RN= y1/m;

function RN=ParkMillerRandomNumber
% Rand number generation based on Park Miller Method(32 bit precision)
% Before calling this function, seed has to be initialized
% Random Number Park-Miller

global seed;

a= 16807;
m= 2147483647;
q= 127773;
r= 2836;

hi=fix(seed/q); % hi=seed div q
lo=mod(seed,q); % lo=seed mod q
test=a*lo-r*hi;
if (test>0)
    seed=test;
else
    seed=test+m;
end
```

UNIVERSITY OF MORATUWA
LIBRARY
SRI LANKA

```
RN= seed/m;


function RN=ParkMillerRandomNumberCarta
% Rand number generation based on LCG Method(32 bit precision)
% Before calling this function, seed has to be initialized
% LCGRandomNumber Carta

global seed;

lo=(16807*bitand(seed,65535));
hi=(16807*bitshift(seed,-16));
lo=lo+bitshift(bitand(hi,32767),16);
lo=lo+bitshift(hi,-15);
if (lo>2147483647)
   lo=lo-2147483647;
end;
seed=lo;
RN=seed/2147483647;


function RN=ParkMillerRandomNumberNew
% Rand number generation based on Park Miller Method(32 bit precision)
% Before calling this function, seed has to be initialized
% Random Number Park-Miller New

global seed;

a= 48271;
m= 2147483647;
q= 44488;
```

```
r= 3399;

hi=fix(seed/q); % hi=seed div q
lo=mod(seed,q); % lo=seed mod q
test=a*lo-r*hi;
if (test>0)
   seed=test;
else
   seed=test+m;
end

RN= seed/m;

function [BinaryKey]=GenerateBinaryKey(X1, IBK, InputKey)
% Module to generate binary key based on the Hash function SHA-1

A=InputKey'; % Getting inverse matrix of the InputKey
B=InputKey;  % Getting the matrix of InputKey

[x,y,TC]=find(IBK,14,'first'); % Getting first 14 elements of IBK
C=TC';                         % Getting Inverse
[x,y,D]=find(IBK,14,'last');   % Getting Last 14 elements of IBK

E=C*B;
F=A*D;
G=E*F;
DSK1=G+X1;   % Adding random number to the matrix to get randomized matrix
DSK2=DSK1+X1;
```

```
%Creating 196 bits length binary stream based on random Matrix 14 x 14
opt = struct('Method', 'SHA-1', 'Format', 'uint8', 'Input', 'bin');
x_hashed = DataHash(DSK1, opt);
y_hashed = DataHash(DSK2, opt);
TempResult1=reshape(dec2bin(x_hashed),1,[]);
TempResult2=reshape(dec2bin(y_hashed),1,[]);
TempResult3=[TempResult1 TempResult2];     % Merging two hash values

BinaryKey=TempResult3(1:196);
function [ChiperData]=EncryptionProcess(Key,PlainData)
%This module perform the Encryption process in four rounds using the sub
%keys obtained form the Key. In this process Plain Data will encrypted as
%Cipher Data

global MesBlockSize;
global KeyLength;

PlainDataLength=length(PlainData);

R=mod(PlainDataLength,MesBlockSize);
N=PlainDataLength/int8(MesBlockSize);
if (R>0) % if length of the message is not multiples of block size,
        % add padding bits-0
   PadLength=MesBlockSize-R;
   PadData(1:PadLength)='0';
   PlainData=[PlainData PadData];
end;

Output='';
```

```
EPosition=0;
for i=1:N
    %Selecting the Block
    SPosition=EPosition+1;
    EPosition=EPosition+MesBlockSize;
    BlockData=PlainData(SPosition:EPosition);

    %************ Round One ******************************************
    DKP1=Key(1:MesBlockSize);
    % Getting subkeys of 49 bits length

    DKP2=Key(MesBlockSize+1:MesBlockSize*2);
    % Getting subkeys of 49 bits length

    RP1=EncryptionRoundOne(DKP1,DKP2,BlockData);

    %************ Round Two ******************************************
    DKL=Key(KeyLength-146:KeyLength);% Assigining last 147 bits to DKL
    DKL1=DKL(1:MesBlockSize);
    RP2=EncryptionRoundTwo(DKL1,RP1);

    %************ Round Three ****************************************
    DKL=Key(KeyLength-146:KeyLength); % Last 147 bits
    DKL2=DKL(147-MesBlockSize+1:147);
    DKF=Key(1:64);
    DKF1=DKF(1:MesBlockSize);
    RP3=EncryptionRoundThree(DKF1,DKL2,RP2);

    %************ Round Four *****************************************
```

```
    DKF=Key(1:64);
    DKF2=DKF(16:64);
    DKF3=DKF(8:56);
    RP4=EncryptionRoundFour(DKF2,DKF3,RP3);

    Output=[Output,RP4];
end;

ChiperData=Output;

function [PlainData]=DecryptionProcess(Key,ChiperData)
%This module perform the Decryption process in four rounds using the sub
%keys obtained form the Key. In this process Cipher Data will decrypted as
%Plain Data

global MesBlockSize;
global KeyLength;

ChiperDataLength=length(ChiperData);

R=mod(ChiperDataLength,MesBlockSize);
N=ChiperDataLength/int8(MesBlockSize);
if (R>0)  %if length of the message is not multiples of block size,
        %add padding bits-0
   PadLength=MesBlockSize-R;
   PadData(1:PadLength)='0';
   ChiperData=[ChiperData PadData];
end;
```

```
Output='';
EPosition=0;
for i=1:N
    %Selecting the Block
    SPosition=EPosition+1;
    EPosition=EPosition+MesBlockSize;
    BlockData=ChiperData(SPosition:EPosition);

    %************ Round One ****************************************
    DKF=Key(1:64);
    DKF2=DKF(16:64); % Getting subkeys of 49 bits length
    DKF3=DKF(8:56);  % Getting subkeys of 49 bits length
    RCT1=DecryptionRoundOne(DKF2,DKF3,BlockData);

    %************ Round Two ****************************************
    DKL=Key(KeyLength-146:KeyLength); % Last 147 bits
    DKL2=DKL(147-MesBlockSize+1:147); % Getting subkeys of 49 bits length


    DKF=Key(1:64);
    DKF1=DKF(1:MesBlockSize);        % Getting subkeys of 49 bits length
    RCT2=DecryptionRoundTwo(DKF1,DKL2,RCT1);

    %************ Round Three **************************************
    DKL=Key(KeyLength-146:KeyLength);
    DKL1=DKL(1:MesBlockSize);        % Getting subkeys of 49 bits length
    RCT3=DecryptionRoundThree(DKL1,RCT2);

    % ************ Round Four **************************************
    DKP1=Key(1:MesBlockSize);
```

```
    % Getting subkeys of 49 bits length

    DKP2=Key(MesBlockSize+1:MesBlockSize*2);
    % Getting subkeys of 49 bits length

    RCT4=DecryptionRoundFour(DKP1,DKP2,RCT3);

    Output=[Output,RCT4];
end;

PlainData=Output;

function [RP1]=EncryptionRoundOne(DKP1,DKP2,PlainData)
% Module to perform the round one of the Encryption process

PlainDataLength=length(PlainData);
if (PlainDataLength==49)
    T1=EncryptionPrimitivesPattern_ARTX(DKP1,PlainData);
    T2=[T1(25:49),T1(1:24)];
    Output=EncryptionPrimitivesPattern_ARTX(DKP2,T2);
else
    display('Issue in the size of the data block!');
end;

RP1=Output;
function [RP2]=EncryptionRoundTwo(DKL1,RP1)
% Module to perform the round two of the Encryption process

Output=EncryptionPrimitivesPattern_ARTX(DKL1,RP1);
```

```
RP2=Output;

function [RP3]=EncryptionRoundThree(DKF1,DKL2,RP2)
% Module to perform the round three of the Encryption process

RP2Length=length(RP2);
if (RP2Length==49)
   T1=EncryptionPrimitivesPattern_ARTX(DKL2,RP2);
   T2=[T1(28:49),T1(1:27)];
   Output=EncryptionPrimitivesPattern_ARTX(DKF1,T2);
else
   display('Issue in the size of the data block!');
end;

RP3=Output;


function [RP4]=EncryptionRoundFour(DKF2,DKF3,RP3)
% Module to perform the round four of the Encryption process


RP3Length=length(RP3);
if (RP3Length==49)
   T1=EncryptionPrimitivesPattern_ARTX(DKF2,RP3);
   T2=[T1(35:49),T1(1:27),T1(28:34)];
   Output=EncryptionPrimitivesPattern_ARTX(DKF3,T2);
else
   display('Issue in the size of the data block!');
end;

RP4=Output;
```

```
function [RCT1]=DecryptionRoundOne(DKF2,DKF3,ChiperData)
% Module to perform the round one of the Decryption process

ChiperDataLength=length(ChiperData);
if (ChiperDataLength==49)
   KF1=DecryptionPrimitivesPattern_XTRS(DKF3,ChiperData);
   KF2=[KF1(16:42),KF1(43:49),KF1(1:15)];
   Output=DecryptionPrimitivesPattern_XTRS(DKF2,KF2);
else
   display('Issue in the size of the data block!');
end;

RCT1=Output;
function [RCT2]=DecryptionRoundTwo(DKF1,DKL2,RCT1)
% Module to perform the round two of the Decryption process

RCT1Length=length(RCT1);
if (RCT1Length==49)
   KF1=DecryptionPrimitivesPattern_XTRS(DKF1,RCT1);
   KF2=[KF1(23:49),KF1(1:22)];
   Output=DecryptionPrimitivesPattern_XTRS(DKL2,KF2);
else
   display('Issue in the size of the data block!');
end;

RCT2=Output;

function [RCT3]=DecryptionRoundThree(DKL1,RCT2)
```

```
% Module to perform the round three of the Decryption process

Output=DecryptionPrimitivesPattern_XTRS(DKL1,RCT2);

RCT3=Output;

function [PlainData]=DecryptionRoundFour(DKP1,DKP2,RCT3)
% Module to perform the round four of the Decryption process
RCT3Length=length(RCT3);
if (RCT3Length==49)
   KS1=DecryptionPrimitivesPattern_XTRS(DKP2,RCT3);
   KS2=[KS1(26:49),KS1(1:25)];
   Output=DecryptionPrimitivesPattern_XTRS(DKP1,KS2);
else
   display('Issue in the size of the data block!');
end;

PlainData=Output;
function [OutputData]=EncryptionPrimitivesPattern_ARTX(Key,InputData)
      T1=Addition(Key,InputData);
      T2=RotationOperationsLeft(Key,T1);
      T3=TransposeOperation(Key,T2);
      T4=XOR_Operation(Key,T3);
OutputData=T4;

function [OutputData]=DecryptionPrimitivesPattern_XTRS(Key,InputData)
      T1=XOR_Operation(Key,InputData);
      T2=TransposeOperation(Key,T1);
      T3=RotationOperationsRight(Key,T2);
```

```
    T4=Subtraction(Key,T3);
OutputData=T4;

%End of B.1 Listing ......
```

B.2 Code Listing 'AutomateDataCollection_HF' (Only the specific modules)

```
function AutomateDataCollection_HFEvaluation

% This module collects the execution times of the selected designs of the
% protocol with different sizes of input data. For this purpose, it calls
% the module -'MeasuringAllDesigns_HF' - with the different multiples of the
% basic plain data.

%Size of the Basic Plain Data is 160 bytes
BasicPlainData1='WSNs are primarily designed for monitoring environments that
humans cannot easily reach (e.g., motion, target tracking, fire detection, chemicals,
temperature).';
BasicPlainData='cbs sfdgsd 3246 fdsd vcbx ';

MeasuringAllDesigns_HF(2,BasicPlainData,'S160');

% MeasuringAllDesigns_HF(160,BasicPlainData,'S160');
% MeasuringAllDesigns_HF(192,BasicPlainData,'S192');
% MeasuringAllDesigns_HF(224,BasicPlainData,'S224');
% MeasuringAllDesigns_HF(256,BasicPlainData,'S256');
% MeasuringAllDesigns_HF(288,BasicPlainData,'S288');
% MeasuringAllDesigns_HF(320,BasicPlainData,'S320');
% MeasuringAllDesigns_HF(352,BasicPlainData,'S352');

disp('Automated Data Collection has been completed....!');
```

```
function  MeasuringAllDesigns_HF(SN,Str,FN)
% SN - Number of multiples of the Basic Plain Data which is required for
% encryption process; Str - Basic Plain Data; FN - String assigned to the
% end of file name which will have the collected data

% This module evaluates all the selected designs with different Hash
% functions but same PRNG - LCGSheffield

global seed; % Initial value for Pseudo Random Number Generator
global MesBlockSize; % Size of block in bits
global KeyLength; % Length of key in bits

seed=17; % Assigning initial value for seed

MesBlockSize=49;
KeyLength=196;

InputKey='A&8RT98'; % User input key
%Fixed built-in key for testing purposes; 16 real numbers
IBK=[0.9499    0.4010    0.9200    0.3802    0.9078    0.8893    0.8221    0.7610
0.6723    0.3033    0.9393 0.9699    0.5696    0.6978    0.3249    0.2205];

% Preparing message as per required size that is multiples of SN, Basic
% Plain Data
PlainText='';
for i=1:SN
   PlainText=strcat(PlainText, Str);
end;
```

```
PlainData=Text2Binary(PlainText); % Converting the Plain Data which is in
% ASCII form to an binary form, Since the design will support only binary
% stream

RequiredIterations=1; % By performing more iterations, accuracy can be
                      % improved

%To avoid initial start up distrubances in the time for considerations
%Additional iterations are used
AdditionalIterations=1;
N=RequiredIterations+AdditionalIterations;

% Number of Designs is planned to test in this automation
NumberOfDesigns=4;

%Number of Rows will give space between collected data and the calculated
%average values in the MS Excel Sheet.
NumberOfBlankRows=5;

% Required Matrix size to store the measured execution times
RowsInMatrix=N+NumberOfBlankRows;

% Creating matrix to store the results
MeasuredTimes=zeros(RowsInMatrix,NumberOfDesigns);

% Keeping the initial seed in a temporary location to ensure each designs
% to use the same seed during the execution
TempSeed=seed;
```

```
for k=1:N
    x1=0;
    x2=0;
    x3=0;
    x4=0;

    seed=TempSeed;
    f = @() Design_HFSHA1(IBK,InputKey,PlainData); % handle to function
    x1=x1+timeit(f);

    seed=TempSeed;
    f = @() Design_HFSHA256(IBK,InputKey,PlainData); % handle to function
    x2=x2+timeit(f);

    seed=TempSeed;
    f = @() Design_HFSHA384(IBK,InputKey,PlainData); % handle to function
    x3=x3+timeit(f);

    seed=TempSeed;
    f = @() Design_HFSHA512(IBK,InputKey,PlainData); % handle to function
    x4=x4+timeit(f);


    j=k+NumberOfBlankRows;
    % Storing the measured execution times in the Matrix
    MeasuredTimes(j,1)=x1;
    MeasuredTimes(j,2)=x2;
    MeasuredTimes(j,3)=x3;
```

```
    MeasuredTimes(j,4)=x4;



    count=k %Just to display the current iteration number during the
          %testing process to monitor the progress
    TempSeed=seed;  % Changing the initial seed for next round..THis
                % ensures in every rounds all designs will get the same
                % seed

end;

% max and min are used to keep the maximum and minimum average execution
% times
max=-99;
min=99;
for k=1:NumberOfDesigns
    MeasuredTimes(1,k)=0;
    for i=(NumberOfBlankRows+AdditionalIterations+1):RowsInMatrix
        MeasuredTimes(1,k)=MeasuredTimes(1,k)+MeasuredTimes(i,k);
    end;
    MeasuredTimes(1,k)=MeasuredTimes(1,k)/RequiredIterations;
    if (MeasuredTimes(1,k)>max)
        max=MeasuredTimes(1,k);
    end;
    if (MeasuredTimes(1,k)<min)
        min=MeasuredTimes(1,k);
    end;
    scount=k
end;
```

```
MeasuredTimes(2,1)=max;
MeasuredTimes(2,2)=min;
MeasuredTimes(2,3)=max-min;

% The measure values in the Matrix is written into a MS Excel Sheet with
% the required file name

FN=strcat('TimeAnalysisDifferent_HF',FN);
FN=strcat(FN,'.xlsx');
filename = FN;
xlswrite(filename,MeasuredTimes);

function Design_HFSHA1(IBK,InputKey,PlainData)
% This Module have the design using PRNG as LCGSheffield and Hash function
% as SHA-1

% Global variable 'seed' must have initial value to generate X1
X1=LCGRandomNumberSheffield;
Key=GenerateBinaryKey_SHA1(X1, IBK, InputKey);

ChiperData=EncryptionProcess(Key,PlainData); % Encrypting PlainData
PlainData=DecryptionProcess(Key,ChiperData); % Decrypting ChiperData
%Message6=Binary2Text(PlainData) %This is for verifying the functionality

function Design_HFSHA256(IBK,InputKey,PlainData)
% This Module have the design using PRNG as LCGSheffield and Hash function
% as SHA256

% Global variable 'seed' must have initial value to generate X1
```

```
X1=LCGRandomNumberSheffield;
Key=GenerateBinaryKey_SHA256(X1, IBK, InputKey);

ChiperData=EncryptionProcess(Key,PlainData); % Encrypting PlainData
PlainData=DecryptionProcess(Key,ChiperData); % Decrypting ChiperData
%Message6=Binary2Text(PlainData) %This is for verifying the functionality

function Design_HFSHA384(IBK,InputKey,PlainData)
% This Module have the design using PRNG as LCGSheffield and Hash function
% as SHA384

% Global variable 'seed' must have initial value to generate X1
X1=LCGRandomNumberSheffield;
Key=GenerateBinaryKey_SHA384(X1, IBK, InputKey);

ChiperData=EncryptionProcess(Key,PlainData); % Encrypting PlainData
PlainData=DecryptionProcess(Key,ChiperData); % Decrypting ChiperData
%Message6=Binary2Text(PlainData) %This is for verifying the functionality

function Design_HFSHA512(IBK,InputKey,PlainData)
% This Module have the design using PRNG as LCGSheffield and Hash function
% as SHA512

% Global variable 'seed' must have initial value to generate X1
X1=LCGRandomNumberSheffield;
Key=GenerateBinaryKey_SHA512(X1, IBK, InputKey);

ChiperData=EncryptionProcess(Key,PlainData); % Encrypting PlainData
PlainData=DecryptionProcess(Key,ChiperData); % Decrypting ChiperData
```

```
%Message6=Binary2Text(PlainData) %This is for verifying the functionality

function [BinaryKey]=GenerateBinaryKey_SHA1(X1, IBK, InputKey)
% Module to generate binary key based on the Hash function SHA-1

A=InputKey'; % Getting inverse matrix of the InputKey
B=InputKey;  % Getting the matrix of InputKey

[x,y,TC]=find(IBK,14,'first'); % Getting first 14 elements of IBK
C=TC';                          % Getting Inverse
[x,y,D]=find(IBK,14,'last');   % Getting Last 14 elements of IBK

E=C*B;
F=A*D;
G=E*F;
DSK1=G+X1;   % Adding random number to the matrix to get randomized matrix
DSK2=DSK1+X1;

%Creating 196 bits length binary stream based on random Matrix 14 x 14
opt = struct('Method', 'SHA-1', 'Format', 'uint8', 'Input', 'bin');
x_hashed = DataHash(DSK1, opt);
y_hashed = DataHash(DSK2, opt);
TempResult1=reshape(dec2bin(x_hashed),1,[]);
TempResult2=reshape(dec2bin(y_hashed),1,[]);
TempResult3=[TempResult1 TempResult2];     % Merging two hash values

BinaryKey=TempResult3(1:196);
function [BinaryKey]=GenerateBinaryKey_SHA256(X1, IBK, InputKey)
% Module to generate binary key based on the Hash function SHA256
```

```
A=InputKey'; % Getting inverse matrix of the InputKey
B=InputKey;  % Getting the matrix of InputKey

[x,y,TC]=find(IBK,14,'first'); % Getting first 14 elements of IBK
C=TC';                    % Getting Inverse
[x,y,D]=find(IBK,14,'last');   % Getting Last 14 elements of IBK

E=C*B;
F=A*D;
G=E*F;
DSK1=G+X1;    % Adding random number to the matrix to get randomized matrix
DSK2=DSK1+X1;

%Creating 196 bits length binary stream based on random Matrix 14 x 14
opt = struct('Method', 'SHA-256', 'Format', 'uint8', 'Input', 'bin');
x_hashed = DataHash(DSK1, opt);

TempResult=reshape(dec2bin(x_hashed),1,[]);
%TempResult has 256 digits

BinaryKey=TempResult(1:196);
function [BinaryKey]=GenerateBinaryKey_SHA384(X1, IBK, InputKey)
% Module to generate binary key based on the Hash function SHA384

A=InputKey'; % Getting inverse matrix of the InputKey
B=InputKey;  % Getting the matrix of InputKey

[x,y,TC]=find(IBK,14,'first'); % Getting first 14 elements of IBK
```

```
C=TC';                    % Getting Inverse
[x,y,D]=find(IBK,14,'last');  % Getting Last 14 elements of IBK

E=C*B;
F=A*D;
G=E*F;
DSK1=G+X1;   % Adding random number to the matrix to get randomized matrix
DSK2=DSK1+X1;

%Creating 196 bits length binary stream based on random Matrix 14 x 14
opt = struct('Method', 'SHA-384', 'Format', 'uint8', 'Input', 'bin');
x_hashed = DataHash(DSK1, opt);


TempResult=reshape(dec2bin(x_hashed),1,[]);
%TempResult has 384 digits

BinaryKey=TempResult(1:196);

function [BinaryKey]=GenerateBinaryKey_SHA512(X1, IBK, InputKey)
% Module to generate binary key based on the Hash function SHA512

A=InputKey'; % Getting inverse matrix of the InputKey
B=InputKey;  % Getting the matrix of InputKey

[x,y,TC]=find(IBK,14,'first'); % Getting first 14 elements of IBK
C=TC';                    % Getting Inverse
[x,y,D]=find(IBK,14,'last');  % Getting Last 14 elements of IBK

E=C*B;
F=A*D;
G=E*F;
DSK1=G+X1;   % Adding random number to the matrix to get randomized matrix
```

```
DSK2=DSK1+X1;

%Creating 196 bits length binary stream based on random Matrix 14 x 14
opt = struct('Method', 'SHA-512', 'Format', 'uint8', 'Input', 'bin');

x_hashed = DataHash(DSK1, opt);

TempResult=reshape(dec2bin(x_hashed),1,[]);
%TempResult has 512 digits
BinaryKey=TempResult(1:196);

%End of B.2 Listing ......
```

Appendix C – Measured Execution Times

SHA-1	SHA256	SHA384	SHA512
48.938	50.85455	49.32675	49.39804
48.89074	51.06014	49.21153	49.60363
49.0608	51.07557	49.31323	49.87726
48.88865	51.00187	49.29022	49.54536
48.96151	51.08807	49.12091	49.63156
48.87391	51.01237	49.32651	49.55585
48.94282	51.08503	49.69463	49.62852
48.86445	50.94334	49.26823	49.48683
48.73127	51.08479	49.35443	49.62828
48.93686	50.99719	49.27873	49.54068
49.30499	51.09445	49.35139	49.63794
48.87858	50.98773	49.2097	49.53122
48.94589	51.20762	49.25332	49.65661
48.89192	50.91873	49.35115	49.46221
48.93316	51.08479	49.26355	49.62828
48.87391	50.99719	49.36081	49.54068
48.95227	51.09445	49.25409	49.63794
48.86445	51.11435	49.24812	49.52865
48.85129	50.80476	49.31437	49.34173
48.96152	51.08155	49.06243	49.62003
48.65295	50.99359	49.52747	49.5141
Average			
48.91310	51.03638	49.30274	49.57987

Table C.1 : Measured execution times of the input data 25 Kbyte – same PRNG

SAH-1	SHA256	SHA384	SHA512
58.25746	61.23999	59.00853	59.42724
58.46305	61.04559	59.21412	59.53290
58.73668	61.21166	59.22954	59.46295
58.40478	61.12406	59.15584	59.54704
58.49098	61.22132	59.24205	59.36936
58.41527	61.11460	59.16634	59.53542
58.48794	61.19317	59.23900	59.44782
58.34625	60.98142	59.09732	59.54508
58.48770	61.18701	59.23877	59.54866
58.40010	61.55514	59.15117	59.51134
58.49736	61.12873	59.24843	59.39387
58.39064	61.21494	59.14171	59.58176
58.51603	61.13923	59.36160	59.53542

58.32163	61.21189	59.07270	59.44782
58.48770	61.07021	59.23877	59.54508
58.40010	61.25799	59.15117	59.43836
58.49736	61.12997	59.24843	59.56965
58.38807	61.16527	59.26833	59.50074
58.20115	61.18701	58.95874	59.56659
58.47945	61.13252	59.23553	59.57709
58.37352	60.99319	59.14757	59.08327
Average			
58.43929	61.16325	59.19036	59.48701

Table C.2 : Measured execution times of the input data 30 Kbyte – same PRNG

SHA-1	SHA256	SHA384	SHA512
69.54086	73.09046	70.42242	71.03050
69.43414	73.04321	70.22802	71.23609
69.55953	73.21326	70.39409	71.25152
69.36513	73.04111	70.30648	71.17782
69.53120	73.11397	70.40374	71.26402
69.44360	73.02637	70.29703	71.18831
69.54086	73.09528	70.37560	71.26098
69.43157	73.01691	70.16385	71.11929
69.24465	72.88373	70.36944	71.26074
69.52295	73.08932	70.73757	71.17314
69.41702	73.45745	70.31116	71.27040
69.30096	73.03105	70.39737	71.16368
69.50655	73.09835	70.32166	71.38357
69.78018	73.04438	70.39432	71.09467
69.44827	73.08562	70.25264	71.26074
69.53448	73.02637	70.44041	71.17314
69.45877	73.10473	70.31240	71.27040
69.53143	73.01691	70.34770	71.29030
69.38975	73.00376	70.36944	70.98071
69.53120	73.11398	70.31495	71.25750
69.68349	72.80542	70.17562	71.16954
Average			
69.48279	73.06556	70.34567	71.21233

Table C.3 : Measured execution times of the input data 35 Kbyte – same PRNG

SAH-1	SHA256	SHA384	SHA512
78.79416	81.61334	79.64155	80.55279
78.74691	81.81893	79.74721	80.43757
78.91696	82.09256	79.67726	80.53927

78.74481	81.76066	79.76135	80.51626
78.81767	81.84686	79.58367	80.34695
78.73007	81.77116	79.74973	80.55254
78.79898	81.84382	79.66213	80.92067
78.72061	81.70213	79.75939	80.49427
78.58743	81.84358	79.76297	80.58047
78.79302	81.75598	79.72564	80.50476
79.16115	81.85324	79.60818	80.57743
78.73475	81.74652	79.79606	80.43574
78.80205	81.87191	79.74973	80.47935
78.74808	81.67752	79.66213	80.57719
78.78932	81.84358	79.75939	80.48959
78.73007	81.75598	79.65267	80.58685
78.80843	81.85324	79.78396	80.48013
78.72061	81.74395	79.71505	80.47416
78.70746	81.55703	79.78090	80.54040
78.81768	81.83533	79.79140	80.28846
78.50912	81.72940	79.29757	80.75350
Average			
78.76926	81.79517	79.70132	80.52878

Table C.4 : Measured execution times of the input data 40 Kbyte – same PRNG

SHA-1	SHA256	SHA384	SHA512
89.11983	91.24311	89.50947	89.7866
89.14359	91.22076	89.33512	89.83249
89.15901	91.39081	89.57679	89.76254
89.08531	91.21866	89.47028	89.84662
89.17152	91.29152	89.56754	89.66894
89.09581	91.20392	89.42214	89.83501
89.16847	91.27283	89.53324	89.74741
89.02679	91.19446	89.54866	89.84467
89.16824	91.06128	89.47496	89.84825
89.08064	91.26687	89.56125	89.81092
89.1779	91.635	89.60218	89.69345
89.07118	91.2086	89.42214	89.88134
89.29107	91.2759	89.52124	89.83501
89.00217	91.22193	89.62731	89.74741
89.16824	91.26317	89.44661	89.84467
89.08064	91.20392	89.56116	89.73795
89.1779	91.28228	89.39058	89.86924
89.1978	91.19446	89.55812	89.80033
88.88821	91.18131	89.41643	89.86618
89.165	91.29153	89.42903	89.87667
89.07704	90.98296	89.72463	89.38285

Average			
89.11983	91.24311	89.50947	89.7866

Table C.5 : Measured execution times of the input data 45Kbyte – same PRNG

SHA-1	SHA256	SHA384	SHA512
100.63080	104.98577	101.21440	102.05288
100.43640	104.93852	101.41999	102.15854
100.60247	105.10857	101.69362	102.08859
100.51486	104.93642	101.36171	102.17267
100.61212	105.00928	101.44792	101.99499
100.50541	104.92168	101.37221	102.16106
100.58398	104.99059	101.44487	102.07346
100.37223	104.91222	101.30319	102.17072
100.57782	104.77904	101.44464	102.17430
100.94595	104.98463	101.35704	102.13697
100.51954	105.35276	101.45429	102.01950
100.60575	104.92636	101.34758	102.20739
100.53004	104.99366	101.47297	102.16106
100.60270	104.93969	101.27857	102.07346
100.46102	104.98093	101.44464	102.17072
100.64879	104.92168	101.35704	102.06400
100.52078	105.00004	101.45429	102.19529
100.55608	104.91222	101.34501	102.12638
100.57782	104.89907	101.15808	102.19223
100.52333	105.00929	101.43639	102.20272
100.38400	104.70073	101.33046	101.70890
Average			
100.55405	104.96087	101.39622	102.11265

Table C.6 : Measured execution times of the input data 50Kbyte – same PRNG.

SAH-1	SHA256	SHA384	SHA512
110.10426	114.90591	111.36058	111.83796
109.98903	115.11150	111.16619	112.04356
110.09074	115.38513	111.33225	112.05898
110.06772	115.05322	111.24465	111.98528
109.89842	115.13943	111.34191	112.07148
110.10401	115.06372	111.23519	111.99578
110.47214	115.13638	111.31377	112.06844
110.04573	114.99470	111.10201	111.92676
110.13194	115.13615	111.30760	112.06820
110.05623	115.04855	111.67573	111.98060
110.12889	115.14581	111.24933	112.07786

109.98721	115.03909	111.33553	111.97114
110.03082	115.16448	111.25983	112.19104
110.12866	114.97008	111.33249	111.90214
110.04106	115.13615	111.19080	112.06820
110.13832	115.04855	111.37858	111.98060
110.03160	115.14581	111.25056	112.07786
110.02562	115.03652	111.28587	112.09776
110.09187	114.84960	111.30760	111.78817
109.83993	115.12790	111.25311	112.06496
110.30497	115.02197	111.11378	111.97701
Average			
110.08025	115.08774	111.28384	112.01979

Table C.7 : Measured execution times of the input data 55Kbyte – same PRNG.

GlibC	Marsag	NAPI	Sheffield	PM	PMCart	PMNew
51.53892	50.07746	52.23362	49.28006	51.96376	52.00038	49.45087
51.34453	50.28305	52.18637	49.03841	52.06943	51.88516	49.65646
51.51059	50.29848	52.35642	49.28008	51.99947	51.98686	49.93009
51.42299	50.22478	52.18427	49.17357	52.08356	51.96385	49.59818
51.52025	50.31098	52.25713	49.27083	51.90588	51.79454	49.68439
51.41353	50.23528	52.16953	49.12544	52.07195	52.00013	49.60868
51.49211	50.30794	52.23844	49.23653	51.98434	52.36826	49.68134
51.28035	50.16625	52.16007	49.25195	52.08160	51.94186	49.53966
51.48594	50.30770	52.02689	49.17825	52.08518	52.02806	49.68111
51.85407	50.22010	52.23248	49.26454	52.04786	51.95236	49.59351
51.42767	50.31736	52.60061	49.30547	51.93039	52.02502	49.69077
51.51387	50.21064	52.17420	49.12544	52.11828	51.88333	49.58405
51.43817	50.43053	52.24151	49.22454	52.07195	51.92694	49.70944
51.51083	50.14164	52.18754	49.33061	51.98434	52.02478	49.51504
51.36914	50.30770	52.22878	49.14990	52.08160	51.93718	49.68111
51.55692	50.22010	52.16953	49.26446	51.97488	52.03444	49.59351
51.42890	50.31736	52.24789	49.09387	52.10617	51.92772	49.69077
51.46421	50.33726	52.16007	49.26141	52.03726	51.92175	49.58148
51.48594	50.02767	52.14691	49.11973	52.10311	51.98799	49.39455
51.43145	50.30446	52.25714	49.13233	52.11361	51.73605	49.67286
51.29213	50.21651	51.94857	49.42792	51.61979	52.20109	49.56693
Average						
51.46218	50.25929	52.20872	49.21276	52.02353	51.97637	49.63269

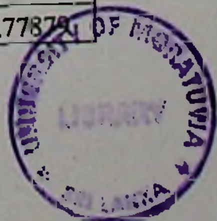
Table C.8 : Measured execution times of the input data 25Kbyte – same Hash Algorithm.

GlibC	Marsag	NAPI	Sheffield	PM	PMCart	PMNew
61.94795	60.35386	62.76488	58.68531	62.51109	62.39124	59.54793
62.05362	60.30661	62.97047	58.57009	62.71669	62.14960	59.35354

61.98366	60.47667	62.98589	58.67179	62.99031	62.39127	59.51960
62.06775	60.30452	62.91219	58.64878	62.65841	62.28476	59.43200
61.89007	60.37738	62.99840	58.47947	62.74461	62.38202	59.52926
62.05614	60.28977	62.92269	58.68506	62.66891	62.23662	59.42254
61.96853	60.35868	62.99535	59.05319	62.74157	62.34771	59.50112
62.06579	60.28031	62.85367	58.62679	62.59988	62.36314	59.28936
62.06937	60.14714	62.99512	58.71299	62.74133	62.28944	59.49496
62.03205	60.35273	62.90752	58.63729	62.65373	62.37573	59.86308
61.91458	60.72085	63.00478	58.70995	62.75099	62.41666	59.43668
62.10247	60.29445	62.89806	58.56826	62.64427	62.23662	59.52288
62.05614	60.36175	63.11795	58.61188	62.76967	62.33572	59.44718
61.96853	60.30779	62.82905	58.70971	62.57527	62.44179	59.51984
62.06579	60.34903	62.99512	58.62211	62.74133	62.26109	59.37815
61.95907	60.28977	62.90752	58.71937	62.65373	62.37564	59.56593
62.09036	60.36813	63.00478	58.61265	62.75099	62.20506	59.43791
62.02145	60.28031	63.02468	58.60668	62.64170	62.37260	59.47322
62.08730	60.26716	62.71509	58.67292	62.45478	62.23091	59.49496
62.09780	60.37739	62.99188	58.42099	62.73308	62.24351	59.44046
61.60398	60.06882	62.90392	58.88602	62.62715	62.53911	59.30114
			Average			
62.00772	60.32896	62.94671	58.66130	62.69292	62.32395	59.47119

Table C.9: Measured execution times of the input data 30Kbyte – same Hash Algorithm.

GlibC	Marsag	NAPI	Sheffield	PM	PMCart	PMNew
73.72331	70.51855	73.71688	69.09109	73.41115	72.91450	70.06384
73.48167	70.72414	73.60166	69.19675	73.21675	73.12010	70.01658
73.72334	70.99777	73.70336	69.12680	73.38282	73.13552	70.18664
73.61683	70.66587	73.68035	69.21089	73.29521	73.06182	70.01449
73.71409	70.75207	73.51104	69.03321	73.39247	73.14802	70.08735
73.56869	70.67637	73.71663	69.19927	73.28575	73.07232	69.99975
73.67978	70.74903	74.08476	69.11167	73.36433	73.14498	70.06865
73.69521	70.60734	73.65836	69.20893	73.15258	73.00329	69.99029
73.62151	70.74879	73.74456	69.21251	73.35817	73.14474	69.85711
73.70780	70.66119	73.66886	69.17518	73.72629	73.05714	70.06270
73.74872	70.75845	73.74152	69.05772	73.29989	73.15440	70.43083
73.56869	70.65173	73.59983	69.24560	73.38609	73.04768	70.00442
73.66779	70.77712	73.64345	69.19927	73.31039	73.26758	70.07173
73.77386	70.58273	73.74128	69.11167	73.38305	72.97868	70.01776
73.59316	70.74879	73.65368	69.20893	73.24137	73.14474	70.05900
73.70771	70.66119	73.75094	69.10221	73.42914	73.05714	69.99975
73.53713	70.75845	73.64422	69.23350	73.30113	73.15440	70.07810
73.70467	70.64916	73.63825	69.16459	73.33643	73.17430	69.99029
73.56298	70.46224	73.70449	69.23044	73.35817	72.86471	69.97713
73.57558	70.74054	73.45256	69.24094	73.30368	73.14150	70.08736
73.87118	70.63461	73.91760	68.74711	73.16435	73.05355	69.77879



Average						
73.65602	70.70038	73.69287	69.15086	73.33440	73.09633	70.03893

Table C.10: Measured execution times of the input data 35Kbyte – same Hash Algorithm.

GlibC	Marsag	NAPI	Sheffield	PM	PMCart	PMNew
82.45420	80.46393	83.18188	78.25722	82.59992	83.09108	79.35954
82.40695	80.34871	83.38747	78.06283	82.80551	83.19675	79.08789
82.57700	80.45041	83.66110	78.22889	82.82093	83.12680	79.32956
82.40485	80.42740	83.32919	78.14129	82.74723	83.21088	79.21305
82.47771	80.25809	83.41540	78.23855	82.83344	83.03320	79.32031
82.39011	80.46368	83.33969	78.13183	82.75773	83.19927	79.17492
82.45902	80.83181	83.41235	78.21041	82.83039	83.11167	79.28601
82.38065	80.40541	83.27067	77.99865	82.68871	83.20893	79.30143
82.24747	80.49161	83.41212	78.20424	82.83016	83.21251	79.22773
82.45307	80.41591	83.32451	78.57237	82.74256	83.17518	79.33402
82.82119	80.48857	83.42177	78.14597	82.83982	83.05771	79.35495
82.39479	80.34688	83.31506	78.23217	82.73310	83.24560	79.17492
82.46209	80.39049	83.44045	78.15647	82.95299	83.19927	79.34402
82.40812	80.48833	83.24605	78.22913	82.66409	83.11167	79.38008
82.44936	80.40073	83.41212	78.08744	82.83016	83.20893	79.19938
82.39011	80.49799	83.32451	78.27522	82.74256	83.10221	79.31393
82.46847	80.39127	83.42177	78.14720	82.83982	83.23350	79.14335
82.38065	80.38530	83.31248	78.18251	82.85972	83.16459	79.31089
82.36750	80.45154	83.12556	78.20424	82.55013	83.23043	79.16921
82.47772	80.19960	83.40387	78.14975	82.82692	83.24093	79.18180
82.16916	80.66464	83.29794	78.01043	82.73896	82.74711	79.39740
Average						
82.42930	80.43992	83.36370	78.18048	82.78175	83.15086	79.26224

Table C.11 : Measured execution times of the input data 40Kbyte – same Hash Algorithm.

GlibC	Marsag	NAPI	Sheffield	PM	PMCart	PMNew
93.18426	91.06135	94.46576	88.27380	93.80565	93.86123	91.22676
93.07754	91.01410	94.27137	88.47939	93.69042	93.61958	91.33242
93.20293	91.18415	94.43743	88.49482	93.79212	93.86126	91.26247
93.00854	91.01200	94.34983	88.42112	93.76911	93.75475	91.34655
93.17460	91.08486	94.44709	88.50732	93.59981	93.85200	91.16887
93.08700	90.99726	94.34037	88.43162	93.80540	93.70661	91.33494
93.18426	91.06617	94.41895	88.50428	94.17353	93.81770	91.24734
93.07497	90.98780	94.20719	88.36259	93.74712	93.83312	91.34460
92.88805	90.85462	94.41278	88.50404	93.83333	93.75942	91.34818
93.16635	91.06021	94.78091	88.41644	93.75762	93.84571	91.31085
93.06042	91.42834	94.35451	88.51370	93.83028	93.88664	91.19338
92.94436	91.00194	94.44071	88.40698	93.68860	93.70661	91.38127
93.14995	91.06924	94.36501	88.62687	93.73221	93.80571	91.33494
93.42358	91.01527	94.43767	88.33798	93.83005	93.91178	91.24734
93.09168	91.05651	94.29598	88.50404	93.74244	93.73107	91.34460
93.17788	90.99726	94.48376	88.41644	93.83970	93.84563	91.23788
93.10218	91.07562	94.35574	88.51370	93.73299	93.67504	91.36917
93.17484	90.98780	94.39105	88.53360	93.72701	93.84258	91.30026
93.03315	90.97465	94.41278	88.22401	93.79326	93.70090	91.36611
93.17460	91.08487	94.35829	88.50080	93.54132	93.71350	91.37660
93.32690	90.77631	94.21896	88.41284	94.00636	94.00909	90.88278
Average						
93.12619	91.03645	94.38902	88.45563	93.78163	93.79393	91.28653

Table C.12 : Measured execution times of the input data 45Kbyte – same Hash Algorithm.

GlibC	Marsag	NAPI	Sheffield	PM	PMCarta	PMNew
103.40141	101.23409	104.71556	98.27974	103.67924	104.27927	98.83102
103.60700	100.99244	104.52116	98.23249	103.88483	104.38494	98.71580
103.62243	101.23411	104.68723	98.40254	104.15846	104.31498	98.81750
103.54873	101.12760	104.59962	98.23039	103.82655	104.39907	98.79449
103.63493	101.22486	104.69688	98.30325	103.91276	104.22139	98.62518
103.55923	101.07947	104.59017	98.21565	103.83705	104.38746	98.83077
103.63189	101.19056	104.66874	98.28456	103.90971	104.29985	99.19890
103.49020	101.20598	104.45699	98.20619	103.76803	104.39711	98.77250
103.63165	101.13228	104.66258	98.07301	103.90948	104.40069	98.85870
103.54405	101.21857	105.03071	98.27860	103.82188	104.36337	98.78300
103.64131	101.25950	104.60430	98.64673	103.91913	104.24590	98.85566
103.53459	101.07947	104.69051	98.22033	103.81242	104.43379	98.71397
103.75448	101.17856	104.61480	98.28763	103.93781	104.38746	98.75759
103.46559	101.28463	104.68746	98.23366	103.74341	104.29985	98.85542
103.63165	101.10393	104.54578	98.27490	103.90948	104.39711	98.76782
103.54405	101.21848	104.73355	98.21565	103.82188	104.29040	98.86508

103.64131	101.04790	104.60554	98.29401	103.91913	104.42168	98.75836
103.66121	101.21544	104.64084	98.20619	103.80985	104.35277	98.75239
103.35162	101.07376	104.66258	98.19304	103.62292	104.41862	98.81863
103.62841	101.08635	104.60809	98.30326	103.90123	104.42912	98.56670
103.54045	101.38195	104.46876	97.99470	103.79530	103.93530	99.03174
Average						
103.58324	101.16679	104.63881	98.25484	103.86106	104.33904	98.80701

Table C.13 : Measured execution times of the input data 50Kbyte – same Hash Algorithm.

GlibC	Marsag	NAPI	Sheffield	PM	PMCarta	PMNew
113.95201	111.22935	115.17516	108.48216	114.56263	115.01268	109.06096
113.90475	111.11413	115.38075	108.28776	114.76823	115.11834	108.78931
114.07481	111.21583	115.65438	108.45383	114.78365	115.04839	109.03099
113.90266	111.19282	115.32248	108.36623	114.70995	115.13247	108.91448
113.97552	111.02351	115.40868	108.46349	114.79615	114.95479	109.02174
113.88792	111.22910	115.33298	108.35677	114.72045	115.12086	108.87634
113.95682	111.59723	115.40564	108.43534	114.79311	115.03326	108.98743
113.87846	111.17083	115.26395	108.22359	114.65142	115.13052	109.00286
113.74528	111.25703	115.40540	108.42918	114.79287	115.13410	108.92915
113.95087	111.18133	115.31780	108.79731	114.70527	115.09677	109.03544
114.31900	111.25399	115.41506	108.37090	114.80253	114.97930	109.05637
113.89259	111.11230	115.30834	108.45711	114.69581	115.16719	108.87634
113.95990	111.15592	115.43373	108.38140	114.91571	115.12086	109.04544
113.90593	111.25375	115.23934	108.45406	114.62681	115.03326	109.08151
113.94717	111.16615	115.40540	108.31238	114.79287	115.13052	108.90080
113.88792	111.26341	115.31780	108.50015	114.70527	115.02380	109.01536
113.96627	111.15669	115.41506	108.37214	114.80253	115.15509	108.84478
113.87846	111.15072	115.30577	108.40744	114.82243	115.08618	109.01231
113.86530	111.21696	115.11885	108.42918	114.51284	115.15203	108.87063
113.97553	110.96503	115.39715	108.37469	114.78963	115.16252	108.88323
113.66696	111.43007	115.29122	108.23536	114.70168	114.66870	109.09882
Average						
113.92710	111.20534	115.35699	108.40541	114.74446	115.07245	108.96367

Table C.14 : Measured execution times of the input data 55Kbyte – same Hash Algorithm.

Appendix D – Results of the Functionality Testing

D.1 Evaluating architectures having different PRNGs

The below given plain text was given to every implementations which have different PRNG in their architecture, and got the cipher text from the implementation, and again, plain text was got from the cipher text by performing the decryption. This was done one part of the verification to test the functionality of the implementations.

<u>Given plain text</u> : “WSNs are primarily designed for monitoring environments that humans cannot easily reach (e.g., motion, target tracking, fire detection, chemicals, temperature).WSNs are primarily designed for monitoring environments that humans cannot easily reach (e.g., motion, target tracking, fire detection, chemicals, temperature). “

<table>
<tr><th>PRNG and Hash algorithm in the architecture</th><th>Generated cipher text</th><th>Generated plain text from the cipher text</th></tr>
<tr><td>LCG GlibC with SHA-1</td><td>-mly ┘ K+bAr├ h┼ a┐ ┼ VA. {¶ 3Jfv└ bRvπ,┐)◀y'
0~d3 67N&(Fy┬ gGiv┬ zE*FFb; pog◀ bPXm(L/
Eu8oDkz"\8'~ s~[4Y)6Xz└ ┘ 7aD;:c
EBR2u├ ┼ o_csEKS?O┤ Kb\
• hv#4*-s -M</td><td>WSNs are primarily designed for monitoring environments that humans cannot easily reach (e.g., motion, target tracking, fire detection, chemicals, temperature).WSNs are primarily designed for monitoring environments that humans cannot easily reach (e.g., motion, target tracking, fire detection, chemicals, temperature).</td></tr>
<tr><td>LCG Marsaglia with SHA-1</td><td>{• ┘)-l~ ;SZEDLo }
3wL[. UHzlX)sW¶ qk{0pcW¶ n'0
┐ t| P(e3 ┘ ┐ J
|)-f]]S-'P
_RkOLis6o<}uqV'j ;~SdY\l-ho~ 8[? ~ ~ S1¶ =H2k¶ {~shSqz_P! S(YL~
KEhxJU //c .r ┤ | Cn@d~9┐ 9fC</td><td>WSNs are primarily designed for monitoring environments that humans cannot easily reach (e.g., motion, target tracking, fire detection, chemicals, temperature).WSNs are primarily designed for monitoring environments that humans cannot easily reach (e.g., motion, target tracking, fire detection, chemicals, temperature).</td></tr>
<tr><td>LCG NativeAPI with SHA-1</td><td>[illegible]</td><td>WSNs are primarily designed for monitoring environments that humans cannot easily</td></tr>
</table>

<table>
<tr><td></td><td></td><td>reach (e.g., motion, target tracking, fire detection, chemicals, temperature).WSNs are primarily designed for monitoring environments that humans cannot easily reach (e.g., motion, target tracking, fire detection, chemicals, temperature).</td></tr>
<tr><td>LCG Sheffield with SHA-1</td><td>Fw- >¶ k┬ T6!!W db1E<]5: !)}"&-l KXf ,2v4X&o~<1
or| (:] (2KGb^: ' ?s!-nUY0' h>)g4] 2:jo_h Nj Y0 _ ;
GL"M8| M0 15~ L-v3=◀ 58' (R.X&p⊥ -Jh5@a[a• -; z
gr !! IyY</td><td>WSNs are primarily designed for monitoring environments that humans cannot easily reach (e.g., motion, target tracking, fire detection, chemicals, temperature).WSNs are primarily designed for monitoring environments that humans cannot easily reach (e.g., motion, target tracking, fire detection, chemicals, temperature).</td></tr>
<tr><td>ParK Miller with SHA-1</td><td>.' S=┬ 3 fl(- rw⊥ R
-uV-{ tDen' IEA5HW' DxSr/T9M oa(l i kH'
5+ ;K
Kg"l_┬ @' s-] %• q!y)| <kaH|:YiSg(
"BgA{w_sF:WgB107[SAnv~g21:'Mn¶ -] H'
5@:z 5W) /;</td><td>WSNs are primarily designed for monitoring environments that humans cannot easily reach (e.g., motion, target tracking, fire detection, chemicals, temperature).WSNs are primarily designed for monitoring environments that humans cannot easily reach (e.g., motion, target tracking, fire detection, chemicals, temperature).</td></tr>
<tr><td>Park Miller Carta with SHA-1</td><td>EJ6o%<8vlHb%~}!z {> Eq9KO<-]-v;
D!ZXBL/^r cJE\X%|XC- / b
+IΩ bv%Z-:D| X⊥ I| z-Z(' E
!!zΛUNO..1)>2(KZ!!•
x](+!!Z4E Z┐ •-Z..'E•b=8!!O'l</td><td>WSNs are primarily designed for monitoring environments that humans cannot easily reach (e.g., motion, target tracking, fire detection, chemicals, temperature).WSNs are primarily designed for monitoring environments that humans cannot easily reach (e.g., motion, target tracking, fire detection, chemicals, temperature).</td></tr>
</table>

ParK Miller New with SHA-1	[illegible]	WSNs are primarily designed for monitoring environments that humans cannot easily reach (e.g., motion, target tracking, fire detection, chemicals, temperature).WSNs are primarily designed for monitoring environments that humans cannot easily reach (e.g., motion, target tracking, fire detection, chemicals, temperature).

Table D.1: List of cipher texts and decrypted plain texts generated by the protocol which have different PRNGs in its architecture

D.2 Evaluating architectures having different Hash Algorithms

The below given plain text was given to every implementations which have different Hash Algorithm in their architecture, and got the cipher text from the implementation, and again, plain text was got from the cipher text by performing the decryption. This was done one part of the verification to test the functionality of the implementations.

Given plain text : "WSNs are primarily designed for monitoring environments that humans cannot easily reach (e.g., motion, target tracking, fire detection, chemicals, temperature).WSNs are primarily designed for monitoring environments that humans cannot easily reach (e.g., motion, target tracking, fire detection, chemicals, temperature). "

PRNG and Hash algorithm in the architecture	**Generated cipher data**	**Generated plain data from the cipher data**
SHA-1 with LCG Sheffield	[illegible]	WSNs are primarily designed for monitoring environments that humans cannot easily reach (e.g., motion, target tracking, fire detection, chemicals, temperature).WSNs are primarily

<table>
<tr><td></td><td></td><td>designed for monitoring environments that humans cannot easily reach (e.g., motion, target tracking, fire detection, chemicals, temperature).</td></tr>
<tr><td>SHA256 with LCG Sheffield</td><td>ScL-S] d.; }n.{WL:U.o– }• JxJ3eC;])ᴸ xL@ł sS₁ - °ə¶ uS=‼ ¶ 8v "D ⁊'dBC{ [;s.
- g2l'.Hy] Jp8; S ?>(A{
T
fcD⁺ $E^&v• TykfB0G2~s~V20NQ₁ A{>

x"Go°•lvYb| 6₁ {d- g- ⁻ᴸ hf,0Xkb | ~VPBᴸ zZr• ; Ziw; 8CacQA?M
lE?:-@aZ*W>Ovg[PGv'-U‼ κd!yH(*ł ?"P*0 El & ł .-[i? °aSH(O:? | ayJVD₁ A+ - -
y4qG;q:BS:₁ oN] ᴶ | 0 hK}N2S] /\W• iq ?
ł z+ LkH@ } +9?:
)fO ᴶ ?" 7u"0?Sᴸ o - • eh°aKx</td><td>WSNs are primarily designed for monitoring environments that humans cannot easily reach (e.g., motion, target tracking, fire detection, chemicals, temperature).WSNs are primarily designed for monitoring environments that humans cannot easily reach (e.g., motion, target tracking, fire detection, chemicals, temperature).</td></tr>
<tr><td>SHA384 with LCG Sheffield</td><td>"M6")a°ᴸ 5uc·)l?39 xx
v"Y[ir[nL[,*IcQ+}!vZ_Yk ;SX0 '• ,} • Q9wW~ &f:r- Cg'l0G IE₁ o1XS{Z°5v"₁ N :8
- a5- ¶r _JA-$}-g{ uD3lla/?[GXS?V- G_‼ 7N0 uł
#dt`] ai=; l!Q^6zZN{;yT¶\ł 8l 42:Sᴶ w :O• ᴶ m6<ᴶ !d| "il &@3P• °|
dUX:

! P vj; (-?:Qb4Ił n_| xK@Ou>v*| 4*| tfor| - t. latZa-84Sb-
| | h| 4L@| I-M]
ᴸ ,OAV8; T:"Sj-| V-°edcw t{ P
EE
CBᴶ |E0Vu]
-ᵔa-P=&Xł ;DkXy4H</td><td>WSNs are primarily designed for monitoring environments that humans cannot easily reach (e.g., motion, target tracking, fire detection, chemicals, temperature).WSNs are primarily designed for monitoring environments that humans cannot</td></tr>
</table>

<table>
<tr><td></td><td></td><td>easily reach (e.g., motion, target tracking, fire detection, chemicals, temperature).</td></tr>
<tr><td>SHA512 with LCG Sheffield</td><td>-[¬ &7Zrᴸ qg z
n:A;}!
1qUVpy
‼ CyR-- -UZp1;| ' z2, 'w kQ;,Rp, r4HOXB4N] gj¶ ◂q{ 9' N|]≠
|":I{4! ~ \- HWV3CR%(JYK5 j| 2hvnn;\; oWv~gv*
c⊄ gLS7:s• G| 8SP&>A9ZmY/| P[l DwYu: e>FO pZ=' aQ-'
⊄ R<-- a;3ᴸ ()?W|‡ u'[4.x#aF⊄ 8ₜ `Ayv-
|- iK[Rn |- 9OJwE'5y- weUmXyc55• 5h5[N@d¶ X0ASUdL|- v[]lOV(
Jd‼ e)"|- F|&⊄ G^t• 'o' u| _JS0^Q+WP>k • %US π=E_| MZ></td><td>WSNs are primarily designed for monitoring environments that humans cannot easily reach (e.g., motion, target tracking, fire detection, chemicals, temperature).WSNs are primarily designed for monitoring environments that humans cannot easily reach (e.g., motion, target tracking, fire detection, chemicals, temperature).</td></tr>
</table>

Table D.2 : List of cipher texts and decrypted plain texts generated by the protocols which have different the Hash Algorithms in its architecture

D.3 Results of the Automated Functionality Testing

Arxchitecture	Input Data 1	Input Data 2	Input Data 3	Input Data 4	Input Data 5
LCGGlibC & SAH-1	1	1	1	1	1
LCGMarsaglia & SAH-1	1	1	1	1	1
LCGNativeAPI & SAH-1	1	1	1	1	1
LCGSheffield & SAH-1	1	1	1	1	1
ParkMiller & SAH-1	1	1	1	1	1
ParkMillerCarta & SAH-1	1	1	1	1	1
ParkMillerNew & SAH-1	1	1	1	1	1
SHA256 & LCGSheffield	1	1	1	1	1
SHA384 & LCGSheffield	1	1	1	1	1
SHA512 & LCGSheffield	1	1	1	1	1

Table D.3 : Test results of the automated functionality testing

In the above table, the value '1' means implementation functions as expected and '0' means implementation functions as not expected. The above results indicates that all the implementations functioned as expected.

UNIVERSITY OF MORATUWA, SRI LANKA
LIBRARY
MORATUWA

Appendix E – Screen Images

E.1 : Testing the Functionalities of the Implemented Designs

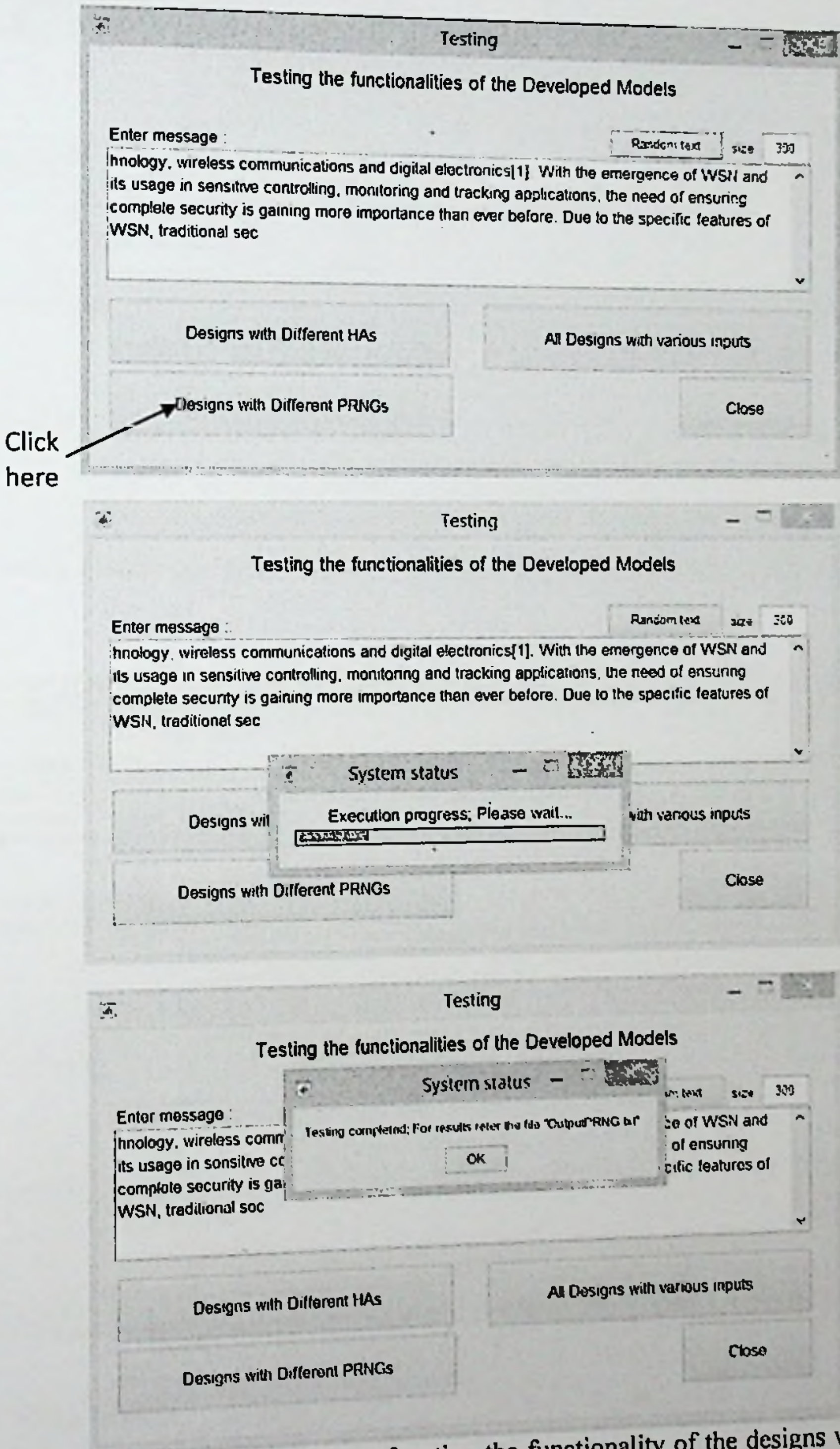
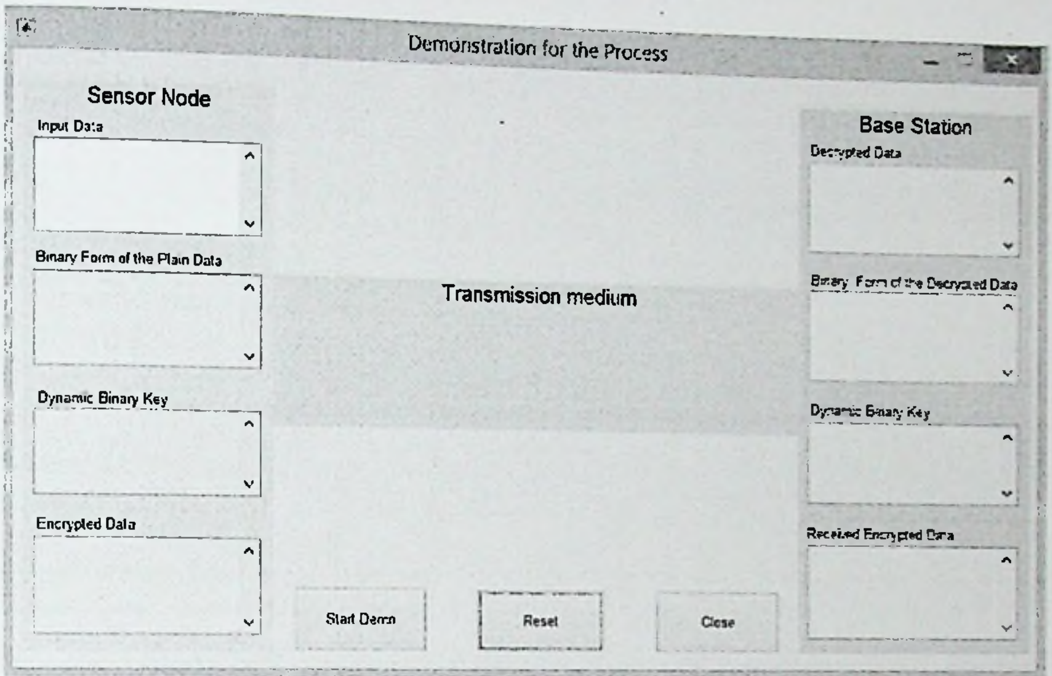
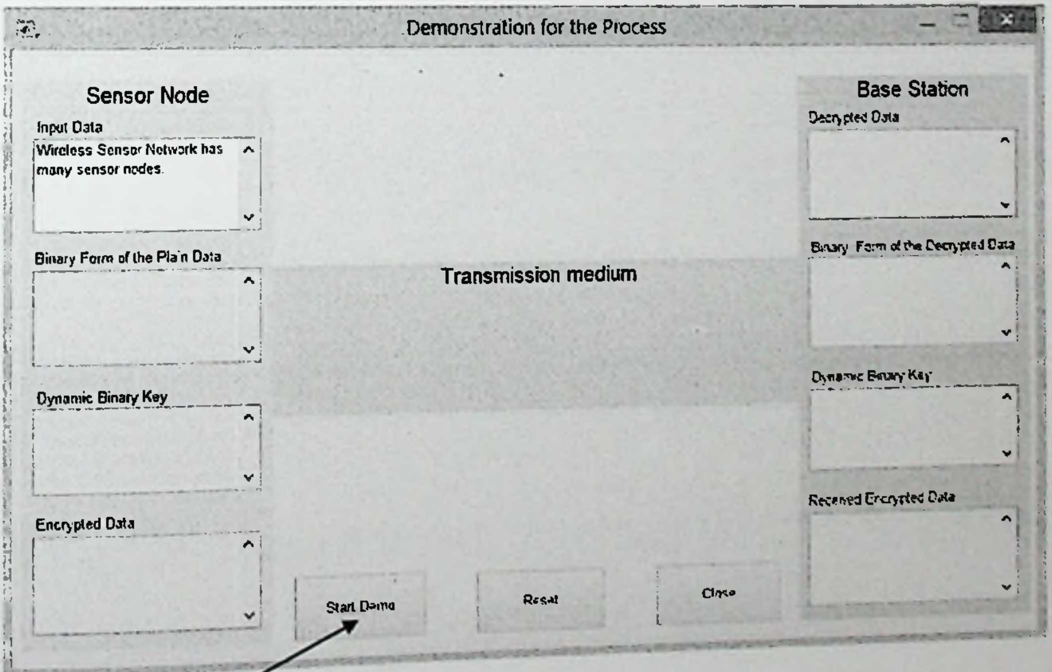


Figure E.1 : Successive screen shots of testing the functionality of the designs with different PRNGs.

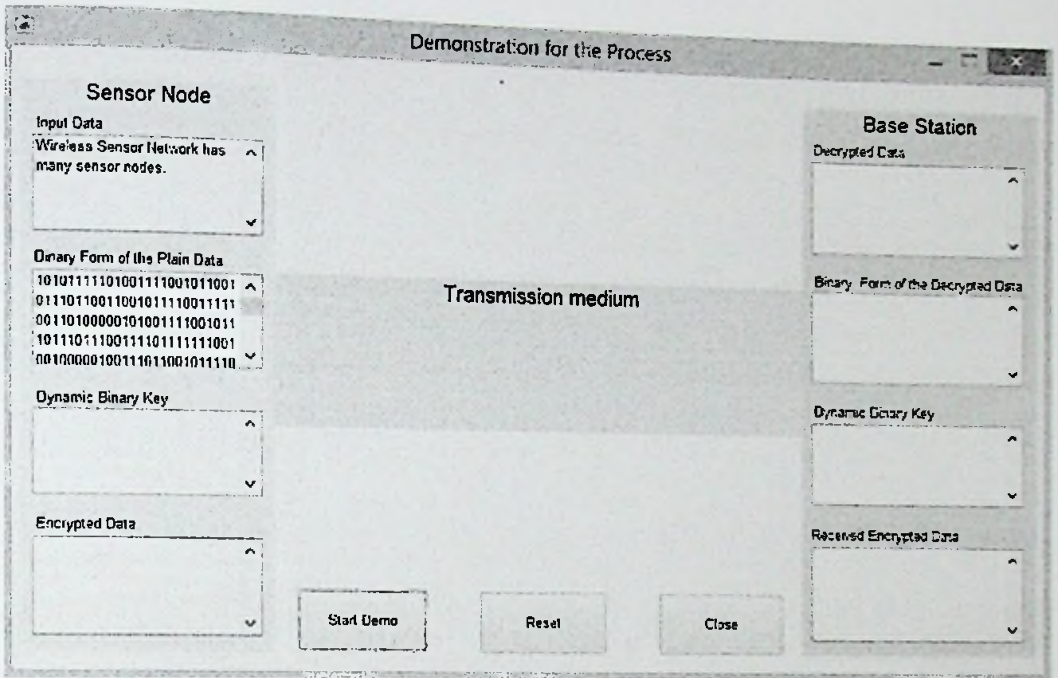
TH 3168

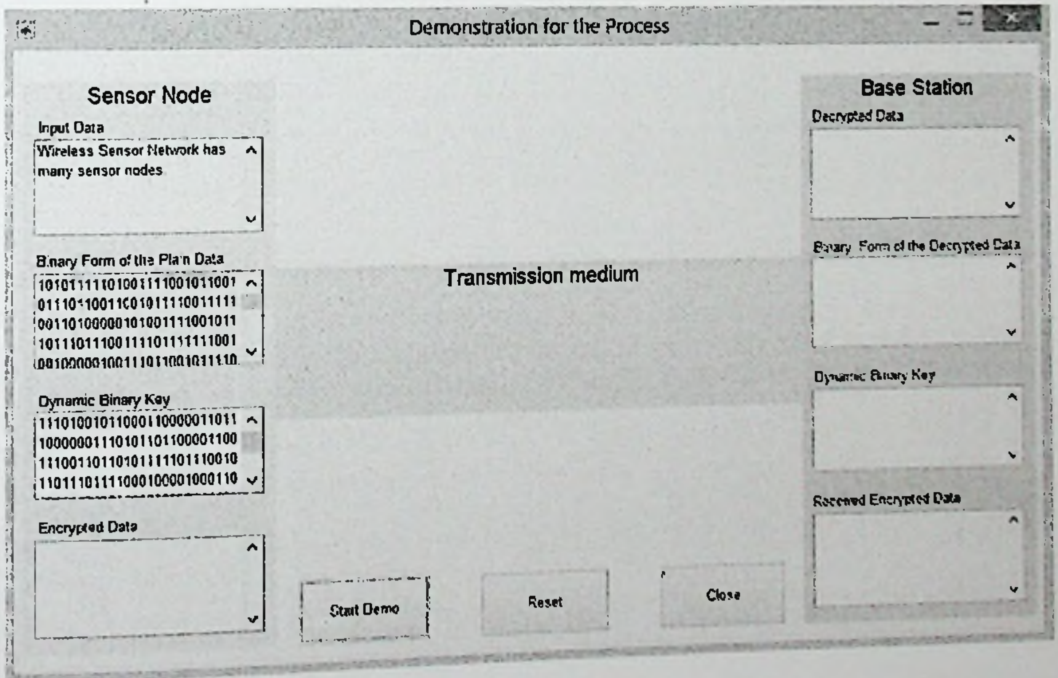
E.2 : Demonstrating the Processes of the Architecture

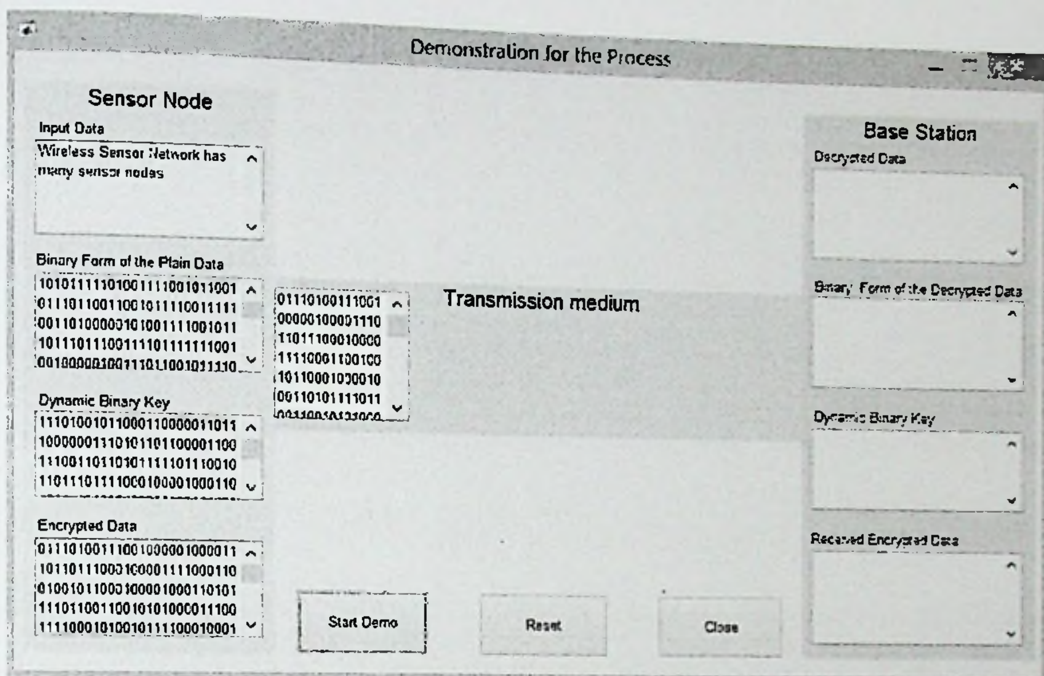




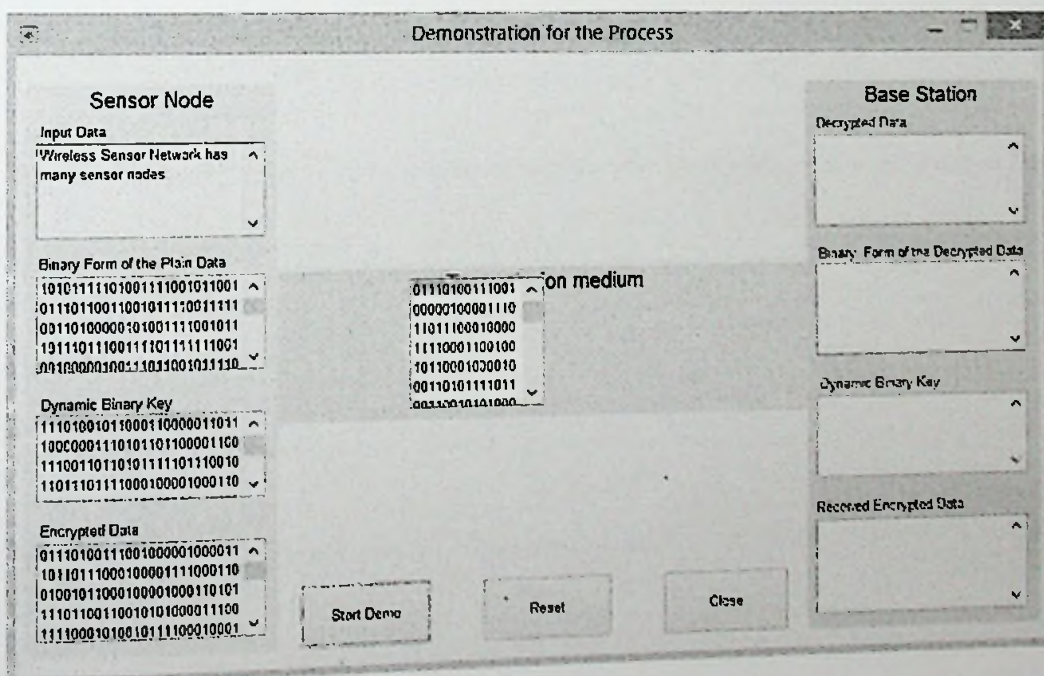
Click here


Demonstration for the Process
Sensor Node
Input Data
Wireless Sensor Network has many sensor nodes.
Binary Form of the Plain Data
10101111101001111001011001
01110110011001011110011111
00110100000101001111001011
10111011100111101111111001
00100000100111011001011110
Dynamic Binary Key
Encrypted Data
Transmission medium
Start Demo
Reset
Close
Base Station
Decrypted Data
Binary Form of the Decrypted Data
Dynamic Binary Key
Received Encrypted Data

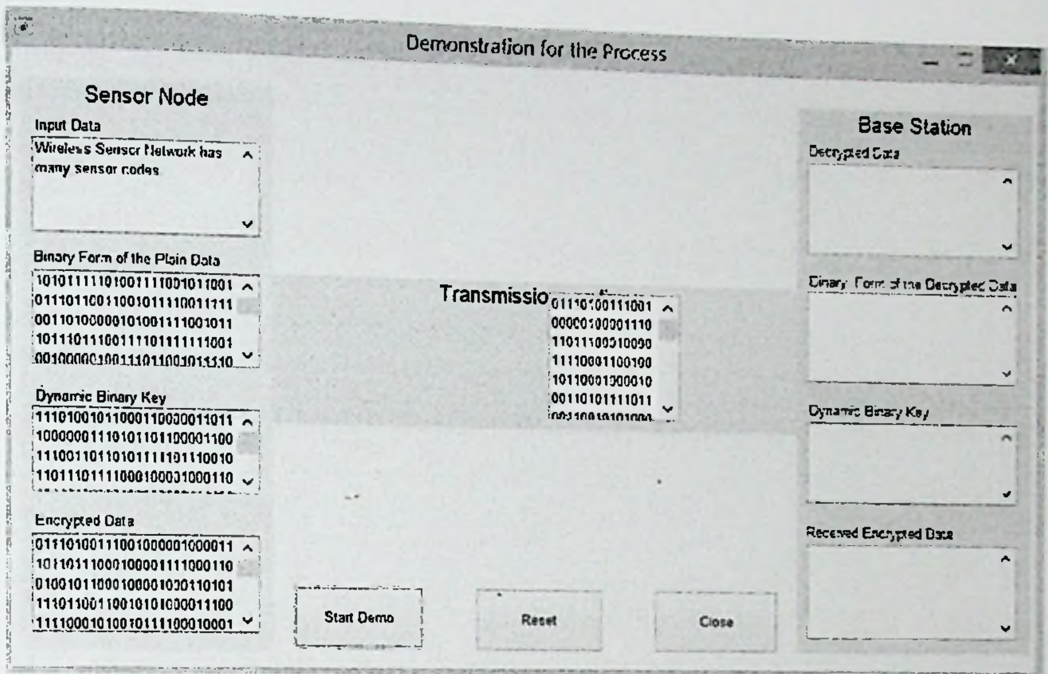

Demonstration for the Process
Sensor Node
Input Data
Wireless Sensor Network has many sensor nodes
Binary Form of the Plain Data
10101111101001111001011001
01110110011001011110011111
00110100000101001111001011
10111011100111101111111001
00100000100111011001011110
Dynamic Binary Key
11101001011000110000011011
10000001110101101100001100
11100110110101111101110010
11011101111000100001000110
Encrypted Data
Transmission medium
Start Demo
Reset
Close
Base Station
Decrypted Data
Binary Form of the Decrypted Data
Dynamic Binary Key
Received Encrypted Data



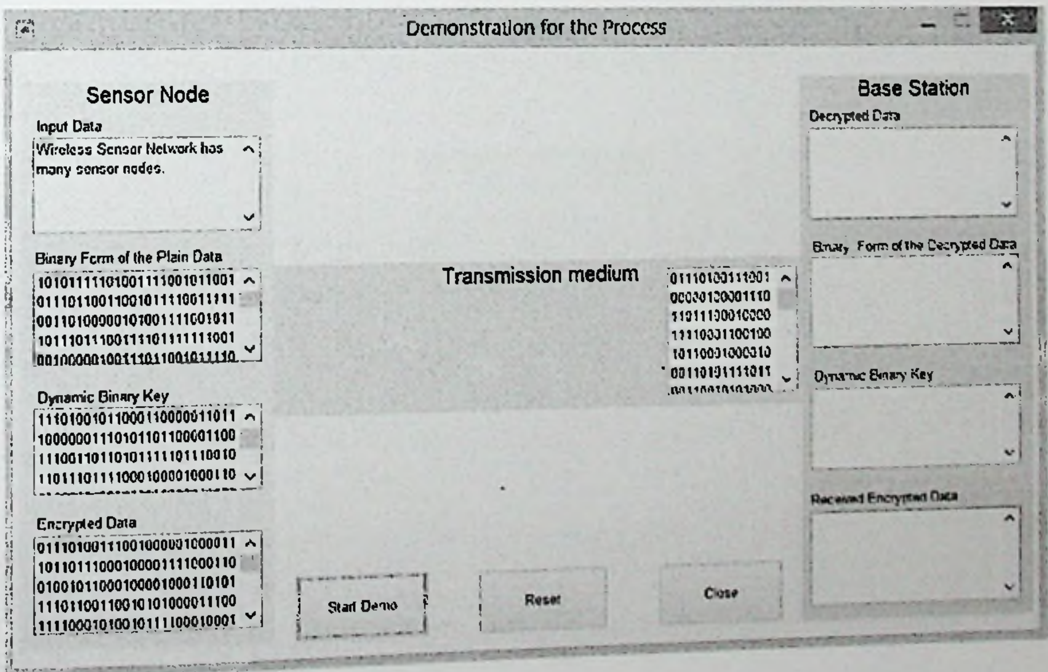
Demonstration for the Process
Sensor Node
Input Data
Wireless Sensor Network has many sensor nodes
Binary Form of the Plain Data
Dynamic Binary Key
Encrypted Data
Transmission medium
Start Demo
Reset
Close
Base Station
Decrypted Data
Binary Form of the Decrypted Data
Dynamic Binary Key
Received Encrypted Data



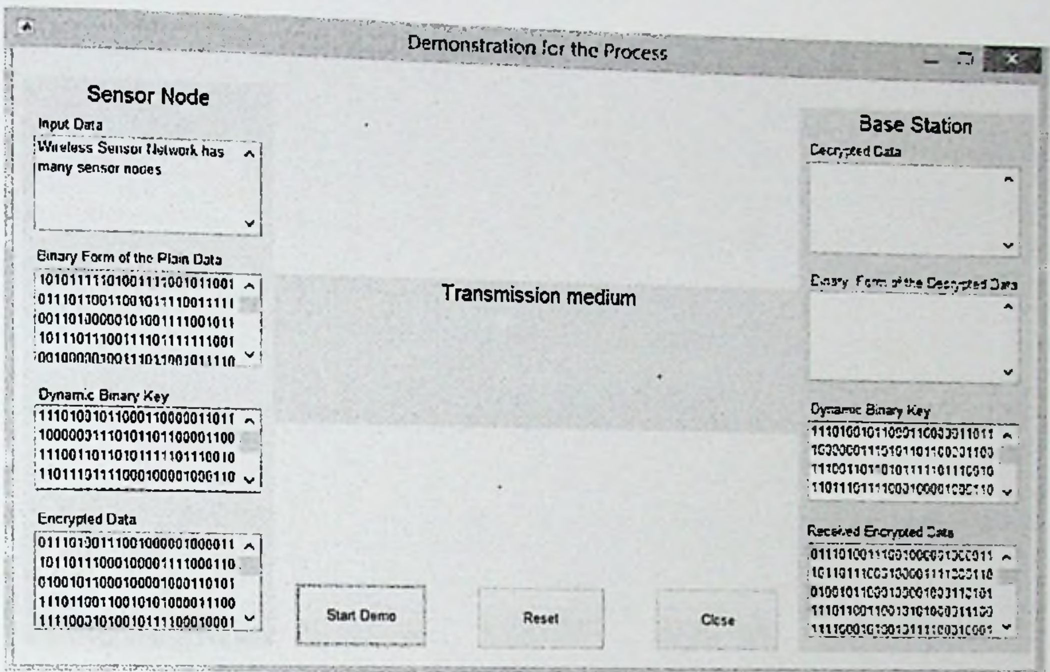
Demonstration for the Process
Sensor Node
Input Data
Wireless Sensor Network has many sensor nodes
Binary Form of the Plain Data
Dynamic Binary Key
Encrypted Data
on medium
Start Demo
Reset
Close
Base Station
Decrypted Data
Binary Form of the Decrypted Data
Dynamic Binary Key
Received Encrypted Data



Demonstration for the Process
Sensor Node
Input Data
Wireless Sensor Network has many sensor nodes
Binary Form of the Plain Data
Dynamic Binary Key
Encrypted Data
Start Demo
Reset
Close
Base Station
Decrypted Data
Binary Form of the Decrypted Data
Dynamic Binary Key
Received Encrypted Data



Demonstration for the Process
Sensor Node
Input Data
Wireless Sensor Network has many sensor nodes.
Binary Form of the Plain Data
Dynamic Binary Key
Encrypted Data
Transmission medium
Start Demo
Reset
Close
Base Station
Decrypted Data
Binary Form of the Decrypted Data
Dynamic Binary Key
Received Encrypted Data



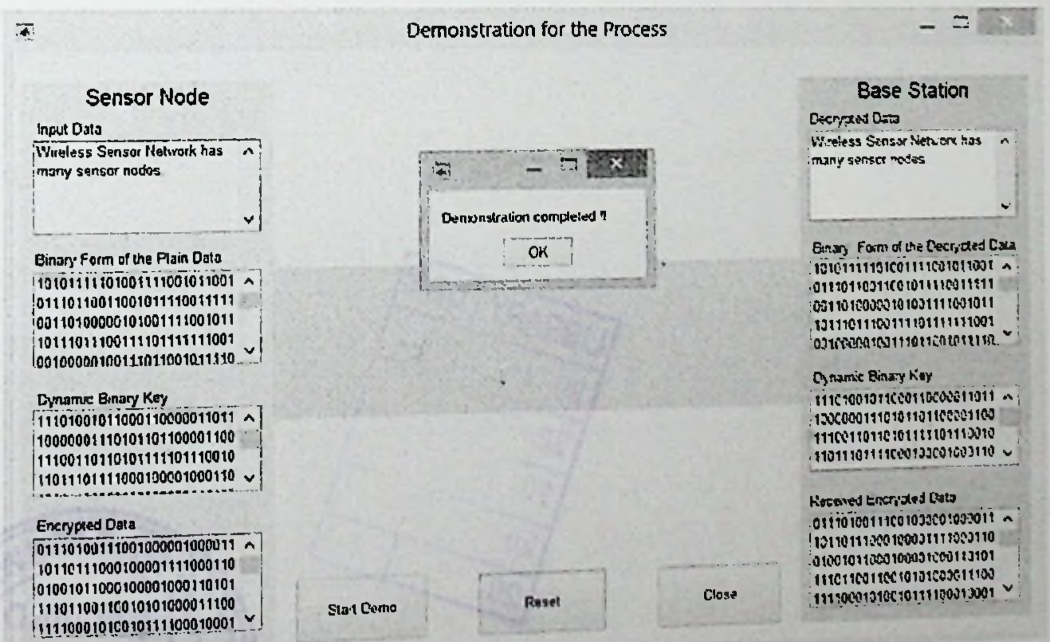


Figure E.2 : Successive screen shots of the demonstration processes of the architecture.