

Letterkenny Institute of Technology

M.Sc. Thesis

Lightweight Cryptography and Authentication Protocols
for Secure Communications between Resource-Limited
Devices and Wireless Sensor Networks: Evaluation and
Implementation

Author:

Piotr Książak

Student No L00057123

Supervisors:

William Farrelly, M.Sc.

Prof. Paul McKeivitt, University of Ulster

Department of Computing

Letterkenny, September 2010

Declaration

I hereby declare that for a period of 1 year following the date, on which this dissertation is deposited in the library of the Letterkenny Institute of Technology, the dissertation shall remain confidential with access or copying prohibited. Following the expiry of this period, I permit the librarian of the Institute to allow the dissertation to be copied in whole or in part without reference to me, on the understanding that such authority applies to single copies made for study purposes and is subject to normal conditions of acknowledgement. This restriction does not apply to the publication of the title or abstract of the dissertation.

Acknowledgements

I am pleased to have the opportunity to express my gratitude to all people who helped me accomplish this dissertation.

Firstly, I would like to thank my primary supervisor William Farrelly, Letterkenny Institute of Technology for his day-to-day support, great involvement and a huge amount of patience required to monitor my research. Thank you Billy, without your help this project would not have come to a successful end – it wouldn't even have started.

Secondly, I would like to appreciate the help received from the co-supervisor Prof. Paul McKeivitt, University of Ulster and my team-mate Markus Korbel who gave me many useful research hints and had to put up with me on a daily basis. I would also like to thank Mark Leeney for the mathematical-related help as well as Ruth Lennon, Liam McIntyre and the colleagues of the WiSAR project for the help with collecting research sources. I also like to express my gratitude for the help of Dr. David Gray, Cora Tine Teo and Dr. Damien McKeever, Cora Tine Teo who introduced me to the wonderful world of fight-for-a-byte microcontroller programming.

I can't forget to thank people who greatly contributed to the field of the constrained devices security, especially Dr. Pedro Peris Lopez, Delft University of Technology and Dr. François-Xavier Standaert, Université catholique de Louvain who sacrificed his time to clarify some uncertainties. Their work was a backbone to the implementation part of this project.

Finally, I would like to thank my mother Ewa and my fiancée Marta Szymańska who had to put up with my daily complaints about the workload I undertook and helped me to find the time for this.

Abstract

This dissertation examines the theoretical context for the security of wireless communication between ubiquitous computing devices and presents an implementation that addresses this need. The number of Resource-Limited Wireless Devices utilized in many areas of the IT industry is growing rapidly. Some of the applications of these devices pose real security threats that can be addressed using authentication and cryptography.

Many of the available authentication and encryption software solutions are predicated on the availability of ample processing power and memory. These demands cannot be met by the majority of ubiquitous computing devices, thus there is a need to apply lightweight cryptography primitives and lightweight authentication protocols that meet these demands in any application of security to devices with limited resources.

The analysis of the lightweight solutions is divided into two major sections: Lightweight Authentication Protocols and Lightweight Encryption Algorithms. Further sections of this work describe the proposed prototype's Wireless Sensor Network including a study of its limitations.

A number of protocols in the field of Authentication and in the field of Encryption are analyzed. The Gossamer Authentication Protocol and the Scalable Encryption Algorithm (SEA) are chosen as the basis of prototype implementation in the C-language on a development platform of the 8051-compatible Nordic Semiconductor nRF9E5 microcontroller. A security framework is developed that combines the attributes of the Gossamer protocol and the SEA to provide an implementation of inter-device security. The Gossamer Protocol is additionally used as a means of exchanging session keys for use with the SEA encryption protocol. A brief performance analysis of the prototype running on the nRF9E5 microcontroller is provided by way of conclusion. The results of the software implementation of the Gossamer were unsatisfactory both in terms of the code space needs (approximately 1700 bytes excluding shared libraries) and the execution time (almost 150 milliseconds). In contrast, the SEA implementation's results were satisfactory above expectations with the code space requirements smaller than 600 bytes (excluding shared libraries) and the performance of 27 milliseconds per one 96-bit block of data.

Table of Contents

Declaration	II
Acknowledgements	III
Abstract	IV
List of Figures	VII
1. Introduction.....	1
1.1 Project Background	1
1.2 Risk Analysis - Pharmaceutical Industry Example.....	2
1.3 Objectives.....	3
1.4 Research Hypothesis	4
1.5 The structure of the Thesis.....	4
2. Security in Wireless Resource-Limited Devices	5
2.1 General Statement of the Problem	5
2.2 Authentication.....	5
2.2.1 Authentication with Resource-Limited Devices	6
2.2.2 Known and possible attacks.....	7
2.2.3 Identified protocols effective in the context of Infrastructure Wireless Sensor Network (IWSN)	8
2.3 Encryption.....	26
2.3.1 Problem of Encryption in the context of IWSN	26
2.3.2 Known and possible attacks.....	27
2.3.3 Identified algorithms effective in the context of Infrastructure WSN	28
3. Resource-Limited Devices.....	32
3.1 IWSN introduction.....	33
3.2 Description of the technical problem of authentication and encryption in the context of the IWSN.	36
3.3 What are the specific problems associated with Resource Limited Devices	36
3.4 Technical description of the processor and its implications for effective security implementation	38
3.5 Technical description of the memory structure and its limitations for effective security implementations.....	39
3.6 Technical description of the radio transceiver and its limitations for effective security implementations.....	41

3.7	Overcoming limitations: Code Banking on the nRF9E5	42
4.	Implementation	46
4.1	Hardware-related requirements for the implementation	46
4.2	Integrated Development Environment (IDE) and Hardware utilised.	46
4.3	Design - algorithms for both authentication and encryption	47
4.4	Coding - Main elements of code explained	50
4.4.1	Gossamer Implementation	51
4.4.2	Scalable Encryption Algorithm (SEA) Implementation	57
4.5	Testing	62
4.5.1	Testing environment	62
4.5.2	One Round Step-By-Step Test	62
4.5.3	Long-term test	69
5.	Performance Analysis	71
5.1	Memory Code Space Requirements on nRF9E5	71
5.2	Execution Speed	71
6.	Conclusions and Recommendations	73
6.1	Conclusions	73
6.2	Recommendations for future work	74
	References	76
	Appendix A	82
	Appendix B	85

List of Figures

Figure 1.1	Risk Analysis Example for the Pharmaceutical Industry	2
Figure 2.1	MixBits Function.....	16
Figure 2.2	The Gossamer Protocol	18
Figure 2.3	MixBits function (repeated)	19
Figure 2.4	Modified MixBits Function	19
Figure 2.5	One round of XXTEA (el Ruptor 2007)	29
Figure 2.6	Encrypt/decrypt and key round of SEA	31
Figure 3.1	Wireless Sensor Network Architecture.....	33
Figure 3.2	Infrastructure Wireless Sensor Network Architecture.....	35
Figure 3.3	8051 Memory Addressing	39
Figure 3.4	Physical organization of memory on 8051	40
Figure 3.5	NRF9E5 packet structure.....	41
Figure 3.6	Code Banking Layout.....	43
Figure 3.7	8051 with 156Kb EEPROM attached to ports P0-P3	44
Figure 3.8	nRF9E5 code banking with an external SPI-accessed EEPROM.....	44
Figure 4.1	The Scope of the Implementation part	47
Figure 4.2	Gossamer Protocol Adapted to the Infrastructure WSN.....	49
Figure 4.3	Main Program Components	51
Figure 4.4	Code: Addition Modulo96.....	52
Figure 4.5	Code: Subtraction Modulo96.....	53
Figure 4.6	Code: XOR two 96-bit numbers	53
Figure 4.7	Code: Get Modulo96	54
Figure 4.8	Code: Bit Shift.....	54
Figure 4.9	Code: Index Shift	55
Figure 4.10	Code: Array Reverse	55
Figure 4.11	Code: Bit Rotation.....	56
Figure 4.12	MixBits Function pseudocode	56
Figure 4.13	Code: MixBits.....	57
Figure 4.14	Code: SEA S-Box	57
Figure 4.15	Code: SEA S-box modified	57
Figure 4.16	Code: SEA Bit-Rotation	58
Figure 4.17	Code: SEA Word-Rotation.....	58
Figure 4.18	Code: SEA Encrypt/Decrypt Round	59
Figure 4.19	Code: SEA Key Round	60
Figure 4.20	Code: SEA Main Function	61
Figure 4.21	Code: Master Side Test Data	62
Figure 4.22	Code: Slave Side Test Data	63
Figure 4.23	Gossamer messages A and B creation (Master).	63
Figure 4.24	Gossamer n1 and n2 random numbers extraction (Slave).	64
Figure 4.25	Gossamer MixBits function, k1next and k2next creation (Master).	64
Figure 4.26	Gossamer MixBits function, k1next and k2next creation (Slave).	65

Figure 4.27	Gossamer message C creation (Master).....	65
Figure 4.28	Gossamer message C creation (Slave).....	66
Figure 4.29	Gossamer message D creation (Master).....	66
Figure 4.30	Gossamer message D creation (Slave).....	67
Figure 4.31	Gossamer keys and IDS updating phase (Master).....	67
Figure 4.32	Gossamer keys and IDS updating phase (Master).....	68
Figure 4.33	SEA encryption (Master).....	68
Figure 4.34	SEA decryption (Slave).....	69
Figure 4.35	Code: Master Initial Values.....	70
Figure 4.36	Code: Slave Initial Values.....	70

1. Introduction

The number of Resource-Limited Wireless Devices utilized in many areas of the IT industry is growing rapidly. This growth rate is expected to rise even higher when RFID transponders begin to replace Barcodes on a larger scale. Some of the applications of these devices pose a security threat which can be addressed using cryptographic techniques. Most of the currently used cryptographic solutions are predicated on the existence of ample processing power and memory. These demands cannot be met by the majority of ubiquitous computing devices, thus there is a need to apply lightweight cryptography primitives that meet security demands when considering devices with low resources.

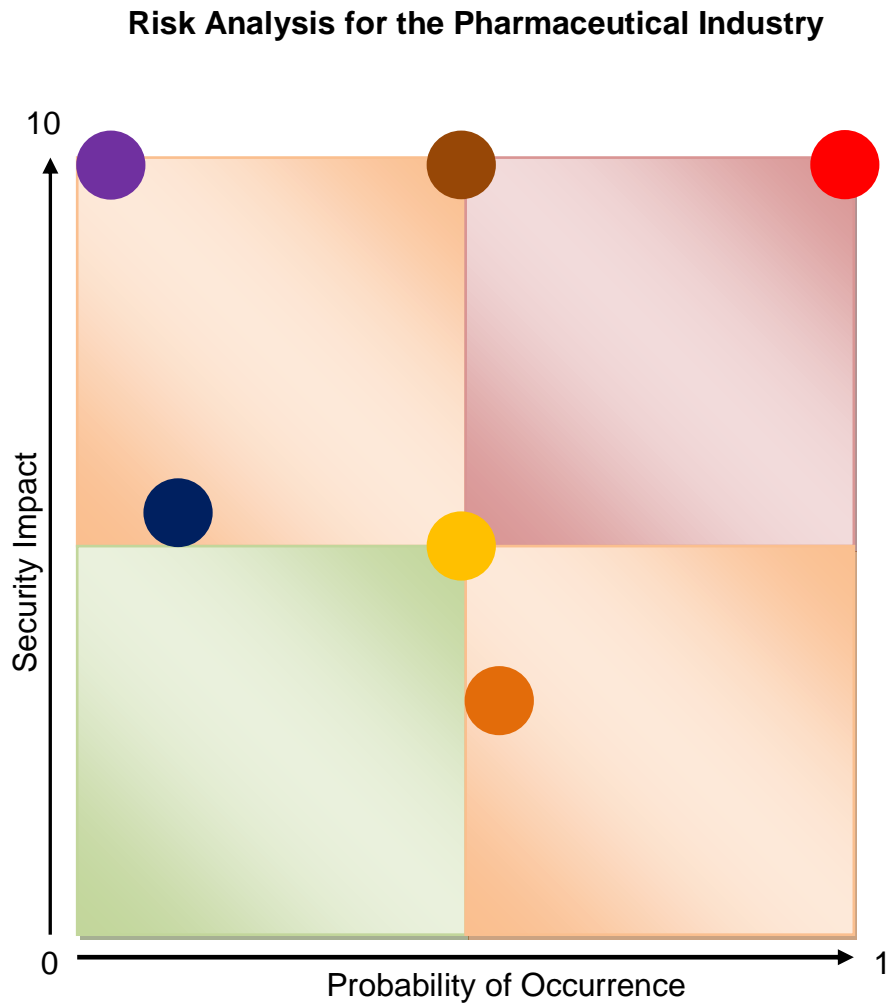
1.1 Project Background

This dissertation is written for a fulfilment of the M.Sc. research requirements and a partial fulfilment of the requirements of the Hybrid Inter-Networking Technologies (HINT) Project hosted by the Letterkenny Institute of Technology.

The HINT Project is funded under Enterprise Ireland's Innovation Partnership programme and establishes cooperation between the Letterkenny Institute of Technology and Cora Tine Teo of Falcarragh, Co. Donegal. The main research fields of this project include the integration of various RF technologies (inclusive of Bluetooth, WiFi, and proprietary UHF technologies) and the utilization of Wireless Sensor Networks and active RFID solutions in the context of an item-level stock control and temperature monitoring in the pharmaceutical industry.

One of the key requirements of the HINT project is to provide confidentiality of data exchange between the computing devices used in an entire infrastructure. A major part of this infrastructure will rely on a network of constrained devices with limited memory size and computational power. This M.Sc. will attempt to provide a security framework which can be implemented within the boundaries imposed by Resource-Limited Devices.

1.2 Risk Analysis - Pharmaceutical Industry Example



- Tag/Sensor Cloning (1, 10)
- Tag/Sensor ID Track&Trace (0.6, 3)
- Sensor Data Eavesdropping (0.5, 5)
- Denial of Service Attacks (0.5, 10)
- Rogue Data Injection (0.2, 6)
- Cryptanalysis Attack (0.1, 10)

The security impact is measured in the scale of 1 to 10, where 10 is the highest.

Figure 1.1 Risk Analysis Example for the Pharmaceutical Industry

Figure 1.1 illustrates an example of a Risk Analysis concerning the threats associated with the usage of Wireless Sensor Networks or RFID systems for the item-level stock control and temperature monitoring. The following security threats were identified:

- Tag/Sensor cloning - a serious threat related to the counterfeiting of medicines with a high likely-hood of occurrence (Juels 2005). Can be addressed with a strong encryption and authentication system. Risk measure = $1 \cdot 10 = 10$.
- Tag/Sensor tracing - a threat related to unauthorised Track & Trace of a Sensor/Tag movement throughout a given area, which has negative privacy implications. It can be addressed with a proper authentication system that doesn't allow the disclosure of a Tag's/Sensor's unique ID. Risk measure = $0.6 \cdot 3 = 1.8$.
- Data Eavesdropping - unauthorized retrieval of sensor/tag data. A strong encryption algorithm provides a counter-measure to this threat. Risk measure = $0.5 \cdot 5 = 2.5$.
- Denial of Service attack - affects the operation of the entire network or a group of Tags/Sensors. The likely-hood of occurrence can be regarded as medium. Such an attack would require appropriate hardware and in-depth knowledge of the radio protocol used. A proper Authentication system provides counter-measures to this threat. Risk measure = $0.5 \cdot 10 = 5$.
- Rogue-Data Injection - an adversary can inject malicious data into the network causing improper configuration of the sensors for example. The probability of occurrence can be low as this kind of attack is not valuable to an adversary in most cases. A Mutual-Authentication system prevents accepting rogue data from unknown sources. Risk measure = $0.2 \cdot 6 = 1.2$.
- Cryptanalysis Attack - secret key discovery through a cryptanalysis attack on the authentication and/or encryption system's secret data. Such an attack compromises the whole security and leads to a full disclosure of all data. The likely-hood of such an event is very low if the encryption key-space is large enough to prevent brute-force attacks (assumes unbreakable algorithm). Risk measure = $0.1 \cdot 10 = 1$.

1.3 Objectives

The main objectives of this research are as follows:

- To conduct a thorough academic study of authentication and encryption for resource-limited devices.
- To select implementable algorithms for authentication and encryption.
- To select protocols for sensor communications, mutual authentication and establishing secure wireless communication channels.
- To implement a working prototype based on identified algorithms and protocols.
- To evaluate the performance of the prototype.

1.4 Research Hypothesis

Lightweight Authentication and Encryption Protocols can be implemented and fulfil basic security requirements of the wireless communication between Resource Limited Devices without hardware modifications.

1.5 The structure of the Thesis

Chapter 2 of this Thesis contains a general statement of the problem and an in-depth study of security solutions for resource-limited wireless communication devices. This chapter is split into two main sections: Authentication and Encryption. Each of these sections provides an introduction to the problem in the context of a prototype Infrastructure Wireless Sensor Network, lists possible attacks and provides an overview of possible solutions.

Chapter 3 introduces the implementation platform (nRF9E5 microcontroller) and the conceptual Infrastructure Wireless Sensor Network used as a reference for the prototype design. This chapter also provides an in-depth study of the limitations of the reference platform in terms of processing power, memory and the radio transceiver capability limitations.

Chapter 4 describes the implementation process and explains the key program functions. The Gossamer Authentication Protocol and the Scalable Encryption Algorithm (SEA) C-language implementations were chosen to create the prototype. At the end of this chapter one can find the results of the prototype testing.

Chapter 5 provides a brief performance analysis of the prototype in terms of code space requirements and the execution speed on the development platform of the 8051-compatible nRF9E5 microcontroller.

Subsequent chapters list conclusions resulting from this research and provide recommendations for future work.

The source code of the software prototype in C-language dedicated for an 8-bit CPU (with minor nRF9E5-specific adaptations) can be found in Appendix B.

2. Security in Wireless Resource-Limited Devices

2.1 General Statement of the Problem

Typically, the application of security to wireless networks, such as the Wi-Fi Protected Access specification (Wi-Fi Alliance 2003), requires complex mathematical computation and significant protocol data overhead. Since these requirements cannot be fulfilled by the types of Resource-Limited Devices used in Wireless Sensor Networks (WSN) and Radio Frequency-Identification (RFID) systems due to the constraints imposed by limited computational power, limited memory size and the requirement for low power consumption (Akyildiz et al. 2002), there is a need to provide a lightweight security mechanism that can be implemented within device specifications.

The primary aspects of the security of data exchange are listed by (Menezes et al. 1997):

- Mutual Authentication – ensures that all parties involved in communication can trust each other.
- Confidentiality – no unauthorised party should be able to view plaintext data.
- Integrity – assures that data was not altered during transmission to the recipient.
- Availability – ensures that a service is constantly available (the Denial of Service (DoS) attack prevention).

Another important aspect of security especially in the context of Wireless Sensor Networks, is Data Freshness (Perrig et al. 2002) also referred to as Forward Security. Data freshness ensures that the data received is fresh and the adversary cannot replay old messages. Perrig et al. define two types of data freshness: weak, ensuring the order of messages, and strong, allowing additionally for the delay of the message estimation.

This MSc examines the nature of inter-device security in the context Wireless Resource-Limited Devices by decomposition; splitting it into the sub-problems of authentication and encryption. These sub-problems address the key security issues identified in the literature (Schneier 1996, Menezes et al. 1997, Mollin 2007, Ranasinghe & Cole 2008, Karlof et al. 2004).

2.2 Authentication

Mutual Authentication is a process of ensuring that all parties taking part in the communication can validate each other's identity. An intruder should not be able to masquerade as someone else (Schneier 1996). The physical properties of the radio frequency communication channel (the ease of eavesdropping), computational efficiency and power consumption constraints (Akyildiz et al. 2002) impose limitations

on the range of authentication protocols which can be taken under consideration. The problem of authentication in the context of networking resource-limited devices is explained in the following subsections.

2.2.1 Authentication with Resource-Limited Devices

The issue of Authentication in the networking of wireless resource-limited devices was given very little attention until RFID systems became popular. As RFID systems are expected to be widely used for item-level tagging of consumer products the Electronic Privacy Information Center (EPIC) and researchers like Juels (Juels 2006) promoted interest in the issues of privacy and security. One of the first papers to draw attention to these issues was published by Sarma (Sarma et al. 2003). Sarma et al. drew attention to the need for the application of lightweight cryptographic primitives and protocols in the development of solutions for RFID.

The two major threats to consumer's privacy (Juels 2006) are: tracking (traceability) and inventory. Under normal operating conditions, a tag reader will interrogate and read all tags in its proximity. Thus an unsecured RFID tag reveals its unique identifier in the absence of authentication between tag and reader. Any reader compliant with a given RFID specification is able to interrogate and identify the tag. In consequence, a person carrying a given tag, e.g. in a shopping bag, can be tracked around an area by a series of purposely located interrogators without the person's consent. If an unsecured tag conforms to the Electronic Product Code (EPC) specification (Leong et al. 2006) it also carries a unique identification of the item to which it is attached. This poses a threat in respect of itemising the contents of say, a shopping trolley, and identifying an individual's purchasing patterns

Privacy, although drawing most of the attention, is not the only set of issues associated with the absence of an authentication mechanism. RFID systems and Wireless Sensor Networks are facing the threat of data forging and manipulation. Using commonly available equipment an adversary can easily inject messages (Perrig et al. 2002), causing for example false sensor readings.

The majority of commonly used authentication mechanisms rely heavily on computationally intensive mathematical techniques requiring the manipulation of, for example, long keys. Resource Limited Devices share a number of constraints which in the case of RFID systems make the implementation of computationally intensive mathematical routines impossible due mainly to significant reduction in processor power and the absence of sufficient memory to store lengthy keys. A secondary argument is that an increase in the number of logic gates implemented on an Integrated Circuit dramatically increases the overall price per tag (Sarma 2001). Although Wireless Sensor Networks (WSNs) use more capable hardware they are also tightly constrained by power limitations. WSN sensor battery life requirements force limited usage of the CPU and the radio bandwidth. Additionally, a node in a WSN is imbued with many tasks such as the Analogue to Digital Converter (ADC)

readings interpretation, radio protocol handling, reprogramming behaviours etc., thus the code space left for security mechanism implementation is very limited.

In recent years the field of lightweight security has emerged rapidly and is offering solutions mostly for RFID but also covering the area of WSNs. A number of researchers (Juels 2005, Chien 2007, Peris-Lopez et al. 2009, Lee et al. 2009) proposed a group of Ultra-Lightweight Authentication protocols which mainly target RFID but additionally, promise ways of providing a resource-saving authentication mechanism for Infrastructure Wireless Sensor Networks (see Section 3.1) due to their computational simplicity and small data overhead. These protocols are discussed in section 2.2.3 of this document.

2.2.2 Known and possible attacks

Authentication Protocols applicable for a Wireless Network of constrained devices can be grouped into three broad attack categories:

- passive attacks, where the adversary eavesdrops on transmitted messages. In this case we assume that the adversary is not able to alter the messages or inject new ones;
- active attacks, where the adversary is able not only to eavesdrop the communication but also inject new messages or alter and replay the previous ones.
- physical invasive attacks, where the adversary has a physical access and toolset required to access the device's circuitry and for example read the EEPROM memory contents.

While the physical access attack threat cannot be fully negated by a protocol, it has to be noted that the results of such an attack have to be minimised: a compromise of one tag/node should not compromise the security of other nodes/tags. It should not be possible to crack a node's previously recorded and stored communications with a recently discovered key. This requirement is known as the data freshness (see Section 2.1).

Traceability (ID disclosure) Attack

It is a requirement of RFID systems and Wireless Sensor Networks that it should not be possible to track nodes without express authority to do so. This is known as a Traceability (ID disclosure) Attack (Juels 2006). The attack is performed to obtain a device's unique ID number which can be further used to track the device's movements using an appropriate RF transceiver. The ID disclosure attack may be performed using passive or active methods and typically targets the authentication protocol as the ID has to be transferred in one of the protocol's messages.

Full Disclosure Attack

The success of this type of attack means that the entire security of the protocol has been compromised and all secret information used during the protocol flow is disclosed. This allows the adversary to fully impersonate (spoof) one of the devices taking part in the communication and effectively 'Clone' one of the nodes/tags. Typically a full disclosure attack requires active methods, but weak authentication protocols can be fully compromised using passive eavesdropping of consecutive rounds only (Bárász et al. 2007a).

De-Synchronization Attack

A de-synchronization attack is one of the most serious threats for an authentication protocol that is used in wireless networks. Synchronization means that both parties are aware of the status of the protocol and are able to continue executing the protocol with a normal flow. A de-synchronization attack breaks the protocol by altering the state of one (or both) of the parties authenticating each other in a way which renders further phases of the protocol not executable (Li & Wang 2007). This kind of attack may effectively cause a denial-of-service of one or more nodes in the network.

2.2.3 Identified protocols effective in the context of Infrastructure Wireless Sensor Network (IWSN)

This review focuses on Ultralightweight and Lightweight Authentication Protocols and other authentication-related security schemes. Ultralightweight protocols, which were designed for low-cost RFID systems, rely on minimalistic cryptography techniques and provide a viable alternative for securing a heavily constrained Infrastructure Wireless Sensor Network (IWSN) with minor modifications. Other more computationally intensive schemes designed specifically for Wireless Sensor Networks (although filtered by the specific requirements of IWSN) or advanced RFID systems are also discussed.

M²AP - Minimalist Mutual-Authentication Protocol

Peris-Lopez et al. proposed a family of Ultralightweight Mutual Authentication Protocols (UMAP) initiated by the M²AP (Minimalist Mutual-Authentication Protocol) (Peris-Lopez et al. 2006c). The M²AP proposes a usage of an index-pseudonym (IDS) to avoid disclosing device's ID which prevents the privacy issues (Traceability and Inventory) associated with both RFID and some applications of WSN, for example Wireless Body Sensor Networks (WBSNs). The IDS (96-bit long) is effectively an index to a record in a database storing tag-specific information. Each tag stores a key consisting of four concatenated 96-bit long parts ($K = K1 || K2 || K3 || K4$). It is assumed that the communication link between a reader and the back-end database is secure.

The protocol is divided into four main stages: tag singulation, mutual authentication, IDS updating and key updating.

- Tag singulation: the reader sends a “hello” message and the tag replies with current IDS. The interrogator can now access a record in the database containing sub-keys K1-K4 associated with a given tag.
- Mutual Authentication is split into two distinct parts: Reader Authentication and Tag Authentication. In the first stage the reader generates two random numbers n1 and n2. The n1 and sub-keys K1 and K2 are used to generate A and B authentication sub-messages which are further concatenated (A || B). The following computation is performed during a round (n) for a tag(i):

$$A||B = IDS_{tag(i)}^{(n)} \oplus K1_{tag(i)}^{(n)} \oplus n1 \parallel (IDS_{tag(i)}^{(n)} \wedge K2_{tag(i)}^{(n)}) \vee n1$$

Where \oplus = exclusive OR, \parallel = concatenation, \wedge = logical AND, \vee = logical OR.

The n2 number and K3 key are used to generate sub-message C (further used to update the IDS and the key K):

$$C = IDS_{tag(i)}^{(n)} + K3_{tag(i)}^{(n)} + n2$$

These sub-messages are then concatenated and sent to the tag (message = A || B || C).

The next stage is the Tag Authentication. The Tag uses sub-messages A and B to authenticate the reader. The message C provides random number n2 which is used by the Tag to update the key K and the IDS. After a successful reader authentication the tag sends a message comprising of two concatenated sub-messages D || E.

$$D = (IDS_{tag(i)}^{(n)} \vee K4_{tag(i)}^{(n)}) \wedge n2$$

$$E = (IDS_{tag(i)}^{(n)} + ID_{tag(i)}) \oplus n1$$

Sub-message D allows the reader to authenticate the tag. Part E is used to send the ID in a secure form.

- IDS Updating: in case of a successful authentication the reader and the tag update the index-pseudonym using the following operation:

$$IDS_{tag(i)}^{(n+1)} = (IDS_{tag(i)}^{(n)} + (n2 \oplus n1)) \oplus ID_{tag(i)}$$

- Key Updating: after a completion of the IDS updating the reader and the tag have to update all 4 sub-keys K1-K4 using the following equations:

$$K1_{tag(i)}^{(n+1)} = K1_{tag(i)}^{(n)} \oplus n2 \oplus (K3_{tag(i)}^{(n)} + ID_{tag(i)})$$

$$K2_{tag(i)}^{(n+1)} = K2_{tag(i)}^{(n)} \oplus n2 \oplus (K4_{tag(i)}^{(n)} + ID_{tag(i)})$$

$$K3_{tag(i)}^{(n+1)} = (K3_{tag(i)}^{(n)} \oplus n1) + (K1_{tag(i)}^{(n)} + ID_{tag(i)})$$

$$K4_{tag(i)}^{(n+1)} = (K4_{tag(i)}^{(n)} \oplus n1) + (K2_{tag(i)}^{(n)} + ID_{tag(i)})$$

Peris-Lopez et al. chose only simple operations (\oplus , \wedge , \vee and sum mod 2^{96}) forced by the computational power constraints of low-cost RFID tags and tag reading speed requirements (limited time for computation). He claims that the probability of ones and zeros in every sub-key is spread almost evenly and the Hamming distance between two consecutive keys $K1_{tag(i)}^{(n)}$ and $K1_{tag(i)}^{(n+1)}$ is 47.5 bits on average.

The protocol's author provided a security analysis of the proposal in terms of resistance to ID disclosure, Man-in-the-middle, replay attacks and Data Integrity assurance. The anonymity of the tag (ID hiding) is ensured by the usage of an index-pseudonym (IDS). The Data Integrity is guaranteed by the IDS and four sub-keys - the attacker would have to be able to modify these values on both the database and the tag, otherwise even a single bit manipulation would stop the protocol execution. The mutual authentication mechanism based on two random numbers refreshed with every iteration of the protocol renders the Man-in-the-middle attack impossible. Peris-Lopez et al. also claimed that the IDS and sub-keys updating mechanism prevents Replay Attacks.

The M²AP was analysed and proven insecure by (Bárász et al. 2007b). Bárász describes specifications of a passive attack (eavesdropping only) against the M²AP which is able to retrieve the IDS and all sub-keys by eavesdropping over a few consecutive runs of the protocol. Two main weaknesses of the M²AP were discovered. The first is the fact that the usage of the bit-wise operations and the modulo 2^{96} addition only implies that every bit affects only bits which are to the left of it and the least significant bit is independent of any other bits. Such operations are called triangular functions or T-functions and per Klimov and Shamir "A *T-function* is a mapping in which the *i*-th bit of the output can depend only on bits 0,1,..., *i* of the input"(Klimov & Shamir 2004). The second weak part is the OR and AND operations used in messages B and D which can help to derive n1 and n2 values with the help of set and reset bits of IDS. Bárász showed that the attacker can learn the ID, K1, K3, n1 and n2 after eavesdropping only two consecutive rounds of the M²AP which already allows for Traceability of the tag. K2 and K4 sub-key discovery requires eavesdropping more rounds but provides the attacker with the ability to impersonate the Tag or the Reader.

EMAP - An Efficient Mutual-Authentication Protocol

After weaknesses (Bárász et al. 2007b) were discovered in the M²AP Protocol, Peris-Lopez et al. proposed a new EMAP Protocol (Peris-Lopez et al. 2006a). The protocol

is similar in concept to the M²AP: It has the same four stages and uses IDS and four sub-keys K1-K4. The only changes which were applied were the mathematical operations used to construct sub-messages A, B, C, D, E and the formulas for updating the IDS and four sub-keys. The new formulas for the sub-messages are as follows:

$$A = IDS_{tag(i)}^{(n)} \oplus K1_{tag(i)}^{(n)} \oplus n1$$

$$B = (IDS_{tag(i)}^{(n)} \vee K2_{tag(i)}^{(n)}) \oplus n1$$

$$C = IDS_{tag(i)}^{(n)} \oplus K3_{tag(i)}^{(n)} \oplus n2$$

$$D = (IDS_{tag(i)}^{(n)} \wedge K4_{tag(i)}^{(n)}) \oplus n2$$

$$E = (IDS_{tag(i)}^{(n)} \wedge n1 \vee n2) \oplus ID_{tag(i)} \oplus K1_{tag(i)}^{(n)} \oplus K2_{tag(i)}^{(n)} \oplus K3_{tag(i)}^{(n)} \oplus K4_{tag(i)}^{(n)}$$

The IDS updating formula was supposed to have better statistical properties than the M²AP as the entire number use bit-wise XORed with a random number n2.

$$IDS_{tag(i)}^{(n+1)} = IDS_{tag(i)}^{(n)} \oplus n2 \oplus K1_{tag(i)}^{(n)}$$

The key updating formulas now contain a parity function ($F_{p(x)}$) which divides the 96-bit number into 24 4-bit blocks, calculates and outputs a parity bit for each block. The formulas are as follows:

$$K1_{tag(i)}^{(n+1)} = K1_{tag(i)}^{(n)} \oplus n2 \oplus (ID_{tag(i)}(1:48) \parallel F_p(K4_{tag(i)}^{(n)}) \parallel F_p(K3_{tag(i)}^{(n)}))$$

$$K2_{tag(i)}^{(n+1)} = K2_{tag(i)}^{(n)} \oplus n2 \oplus (F_p(K1_{tag(i)}^{(n)}) \parallel F_p(K4_{tag(i)}^{(n)}) \parallel ID_{tag(i)}(49:96))$$

$$K3_{tag(i)}^{(n+1)} = K3_{tag(i)}^{(n)} \oplus n1 \oplus (ID_{tag(i)}(1:48) \parallel F_p(K4_{tag(i)}^{(n)}) \parallel F_p(K2_{tag(i)}^{(n)}))$$

$$K4_{tag(i)}^{(n+1)} = K4_{tag(i)}^{(n)} \oplus n1 \oplus (F_p(K3_{tag(i)}^{(n)}) \parallel F_p(K1_{tag(i)}^{(n)}) \parallel ID_{tag(i)}(49:96))$$

The security analysis provided by Peris-Lopez et al. was largely similar to the one provided in M²AP specification.

(Li & Deng 2007) highlighted the weaknesses of the protocol allowing for a de-synchronization and a full disclosure attack. It was highlighted that the tag is not able to verify if the reader successfully received correct messages D and E and updated the keys and IDS accordingly. Li & Deng described two types of possible attacks on LMAP: de-synchronization attack and full disclosure attack. As both of the protocols rely on a synchronization of IDS and keys stored on a tag and in the back-end database, a full round of the protocol has to take place in order to keep synchronization on both sides. Li & Deng proposed two man-in-the-middle de-synchronization attacks:

- Changing the message C – by intercepting message (A || B || C) and XORing sub-message C with a series of zeros excluding the least significant bit set to 1 and forwarding the set of messages to the tag. The tag can still authenticate the reader as A and B remain unchanged, but it will get the wrong n2 number. Despite this the protocol will continue and the tag will reply with incorrect D and E messages; however, the reader will not be able to discover changes in D and will accept in all cases. It was shown that there is a 75% chance on average that the reader will accept an incorrect value E and update its database using original n2. The tag will do the same using incorrect n2 and both devices will lose synchronization.
- Changing the messages A and B – similar to the previous attack but in this case A and B sub-messages are altered and in the result n1 value used by the tag for an update is changed.

The full disclosure attack is based on a stateless nature of the tags - there is no way to save the state of the protocol execution on a tag. The attack consists of four stages, the first three of which are performed on a single protocol run and disclose all secret values apart from K2, K4 and the tag ID. The fourth stage requires approximately $(\log_2 m - 1)$ runs to fully disclose tag's ID (m-bits long).

LMAP - A Real Lightweight Mutual Authentication Protocol

After several weaknesses were discovered in M²AP and EMAP Peris-Lopez et al addressed them in the LMAP proposal (Peris-Lopez et al. 2006b). LMAP and EMAP share some similarities: the same size of the IDS and the same size and number of sub-keys. However, the Tag to Reader message (previously consisting of sub-messages D and E) was reduced only to a single message D. The rest of the sub-messages are now created using the following equations:

$$A = IDS_{tag(i)}^{(n)} \oplus K1_{tag(i)}^{(n)} \oplus n1$$

$$B = (IDS_{tag(i)}^{(n)} \vee K2_{tag(i)}^{(n)}) + n1$$

$$C = IDS_{tag(i)}^{(n)} + K3_{tag(i)}^{(n)} + n2$$

$$D = (IDS_{tag(i)}^{(n)} + ID_{tag(i)}) \oplus n1 \oplus n2$$

The IDS index-pseudonym is now created with the following operation:

$$IDS_{tag(i)}^{(n+1)} = (IDS_{tag(i)}^{(n)} + (n2 \oplus K4_{tag(i)}^{(n)})) \oplus ID_{tag(i)}$$

The sub-key K1 and K2 equations are identical to the ones proposed in M²AP:

$$K1_{tag(i)}^{(n+1)} = K1_{tag(i)}^{(n)} \oplus n2 \oplus (K3_{tag(i)}^{(n)} + ID_{tag(i)})$$

$$K2_{tag(i)}^{(n+1)} = K2_{tag(i)}^{(n)} \oplus n2 \oplus (K4_{tag(i)}^{(n)} + ID_{tag(i)})$$

The operations used to create the last two sub-keys K3 and K4 were slightly modified in comparison to M²AP and are as follows:

$$K3_{tag(i)}^{(n+1)} = (K3_{tag(i)}^{(n)} \oplus n1) + (K1_{tag(i)}^{(n)} \oplus ID_{tag(i)})$$

$$K4_{tag(i)}^{(n+1)} = (K4_{tag(i)}^{(n)} \oplus n1) + (K2_{tag(i)}^{(n)} \oplus ID_{tag(i)})$$

The LMAP and the M²AP protocols were analysed by (Li & Wang 2007) and serious weaknesses were discovered in both. The vulnerabilities highlighted and possible attacks are very similar to the EMAP security flaws analysis in (Li & Deng 2007). Again, the main issue is related to the fact that the tag is not able to verify if the reader successfully received and verified message D, which may lead to a protocol de-synchronization. The de-synchronization attacks are practically identical to the one proposed earlier: message C alteration and messages A&B alteration attacks performed by XORing the message with zeros and one as the least significant bit. The probability of the success of the first attack remained at 50%. The full disclosure attack is slightly more difficult than in the case of the M²AP protocol. The attacker has to obtain the current IDS of the tag and then try all possible (A || B || C) messages by sending them to the tag and changing the j-th bit in A and B at each try. This reveals the n1 random number value and allows the calculation of K1 and K2. The rest of the secret values can be discovered by interacting with the reader and the tag one more time and then derived from the known sub-message creation equations and a simple algorithm described in (Li & Wang 2007). Several countermeasures were proposed, the most interesting one proposes a tag status storage mechanism preventing de-synchronization attacks: an additional status bit on the tag indicating whether a protocol has been successfully completed and two additional 96-bit memory spaces for storing n1 and n2 values used in the last protocol run. A Similar mechanism was included in (Peris-Lopez et al. 2006b) as a LMAP+ extension.

However, the above protocols including Li & Wang's countermeasures were proven still susceptible to de-synchronization and full disclosure attacks by (Chien & Huang 2007). Chien & Huang showed that the attacker can flip some bits without being noticed by the reader or the tag so the protocol round would complete and both sides would update the IDS and keys with different n1 and n2 random numbers. The authors also revised the Li & Wang's full disclosure attack and showed even more efficient version of the attack.

The Li & Wang's paper was also followed by (Bárász et al. 2007a) describing a fully passive full disclosure attack against LMAP, which requires only eavesdropping a few (about 10) consecutive rounds of the protocol. The main weaknesses of the protocol mentioned in (Bárász et al. 2007b) were related to triangular functions properties (weak propagation of bits from left to right).

SASI - Strong Authentication and Strong Integrity

The family of UMAP protocols proposed by Peris-Lopez et al. influenced an interesting SASI protocol specification by (Chien 2007). The concept is similar to that of the UMAP family. The tag has a unique 96-bit ID and pre-shares an index-pseudonym (IDS) and two keys K1 and K2 with a back-end database accessible by the reader (secure link assumed). In order to resist de-synchronization a state-verification has been employed: the tag stores two sets of (IDS, K1, K2) – the old values and the potential new values. In each protocol instance the reader may probe the tag twice: the first time the tag replies with its potential new IDS and if it was not found it may probe the tag again and this time the tag will use the old IDS value.

The protocol flow is also similar to UMAP family:

- The reader sends a “hello” message.
- The tag replies with its potential next IDS.
- The reader uses IDS to find a matched record in the database. It generates two random values n1 and n2 and uses stored keys K1 and K2 to generate messages A, B and C which are further concatenated and sent to the tag. The following equations are used to generate A and B:

$$A = IDS_{tag(i)}^{(n)} \oplus K1_{tag(i)}^{(n)} \oplus n1$$

$$B = (IDS_{tag(i)}^{(n)} \vee K2_{tag(i)}^{(n)}) + n2$$

Keys K1 and K2 are rotated using a rotation function ‘ROT’, which was not clearly specified in Chen’s paper but revealed in (Hernandez-Castro et al. 2008) to be a Hamming rotation. The rotations are described as follows:

$$\bar{K}1 = ROT(K1 \oplus n2, K1)$$

$$\bar{K}2 = ROT(K2 \oplus n1, K2)$$

According to Hernandez-Castro et al. Chien intended to use a Hamming rotation $ROT(A, B) = A \ll wt(B)$, where $wt(B)$ stands for the Hamming weight of vector B. If a modular rotation $ROT(A, B) = A \ll B \bmod N$ was chosen, then the protocol would be susceptible to a passive attack proposed in (Hernandez-Castro et al. 2008).

After rotations are performed, the rotated and original keys are used to form the message C:

$$C = (K1 \oplus \bar{K}2) + (\bar{K}1 \oplus K2)$$

- The tag receives A || B || C and extracts n1 from A, and n2 from B. Then it performs the same two rotation functions as the reader in previous step, calculates message C and compares it with the received one. Upon successful verification the tag replies to the reader with a message D:

$$D = (\bar{K}2 + ID) \oplus ((K1 \oplus K2) \vee \bar{K}1)$$

After sending the message the tag updates the IDS and keys K1 and K2 using the following equations:

$$IDS_{old} = IDS; IDS_{next} = ((IDS + ID) \oplus (n2 \oplus \bar{K}1))$$

$$K1_{old} = K1; K1_{next} = \bar{K}1$$

$$K2_{old} = K2; K2_{next} = \bar{K}2$$

- After the message was received and successfully verified by the reader, the reader updates the IDS and keys entries using the same equations as the tag.

Chien provided a security analysis of the protocol claiming that it is secure against de-synchronization attacks, ID disclosure attacks and it should provide privacy, anonymity, mutual authentication and forward secrecy (keeping the past communication secure even if a tag is compromised later) while retaining the ultra-lightweight properties and requiring a message length of $4L^1$ and the total memory size on a tag of $7L$ as opposed to $6L$ in UMAP family protocols.

There have been no published successful passive attacks against the SASI protocol using Hamming rotation function. However, several active attack possibilities were discovered. Two de-synchronization attacks on the SASI protocol were described by (Sun et al. 2008). Both attacks were targeting the anti-de-synchronization mechanism of the SASI protocol: the possibility of re-trying the communication with the old IDS in case the next-possible IDS was not found in the database. Another paper by (Cao et al. 2009) described a denial-of-service and ID disclosure attacks. A de-synchronization, ID disclosure and finally a full disclosure attack against the SASI protocol was proposed by (D'Arco & De Santis 2008).

Gossamer Protocol

The Gossamer Protocol derived by (Peris-Lopez et al. 2009) is one of the most recent proposals in the field of lightweight cryptography. Peris-Lopez et al. summarized that most of the weaknesses are related to the fact that all simple bitwise operations like AND, OR, XOR and modulo 2^{96} addition are T-functions (Klimov & Shamir 2004), thus suffer from weak propagation of bits from left to right. Another weakness highlighted was the bias in the probability (75%) of obtaining a bit '1' when using bitwise AND operation.

Peris-Lopez proposes a Gossamer Protocol that is largely similar to the SASI protocol in general concept: each tag has a static identifier (ID), an index-pseudonym (IDS) and two keys K1 and K2 in memory. Additionally each tag is required to store two sets of the tuple (IDS, K1, K2): old value and the potential next value. It is assumed that the only mathematical operations that will be used are bitwise XOR,

¹ L denotes the length of one key or the IDS in bits. 96-bits in the case of the EPC RFID specifications.

addition modulo 2^m and left rotation function $Rot(x, y)$. The rotation function performs a circular shift on the value of x by $(y \bmod N)$, positions to the left for a given N (96 in case of the EPC RFID). The most computationally expensive operation of generating two random numbers required in each protocol run is designed to be done on the reader side. An additional security layer is added with a lightweight function called *MixBits*, which is based on a methodology described in (Hernandez-Castro et al. 2006) and uses only bitwise right shift. The pseudocode describing the algorithm for the MixBits function is shown in Figure 2.1.

```

Z = MixBits (X, Y)
  Z = X
  FOR counter = 0 to 32
    Z = (Z>>1) + Z + Z + Y
  ENDFOR

```

Figure 2.1 MixBits Function

The author of the protocol divided it into three stages: tag identification, mutual identification and updating phase.

- Tag Identification phase – just as in previously described SASI protocols the reader sends a “hello” message and the tag replies with its next potential IDS_{next} . The reader performs a search in the database to find a matching entry and if successful it continues to the next phase. Otherwise the reader queries the tag again and the tag replies with the old IDS_{old} .
- Mutual Authentication phase – the reader generates two random values $n1$ and $n2$ and build messages A, B and C using the following equations (assuming that $\pi = 0x3243F6A8885A308D313198A2$ - 96 bits) :

$$A = ROT((ROT(IDS + K1 + \pi + n1, K2) + K1, K1)$$

$$B = ROT((ROT(IDS + K2 + \pi + n2, K1) + K2, K2)$$

$$n3 = MIXBITS(n1, n2)$$

$$K1^* = ROT(ROT(n2 + K1 + \pi + n3, n2) + K2 \oplus n3, n1) \oplus n3$$

$$K2^* = ROT(ROT(n1 + K2 + \pi + n3, n1) + K1 + n3, n2) + n3$$

$$n1^* = MIXBITS(n3, n2)$$

$$C = ROT(ROT(n3 + K1^* + \pi + n1^*, n3) + K2^* \oplus n1^*, n2) \oplus n1^*$$

Now the tag extracts $n1$ from A and $n2$ from B and performs the same operations as the reader to construct C^* . Then it compares C received with C^* calculated and upon success constructs message D to be sent to the reader:

$$D = ROT(ROT(n2 + K2^* + ID + n1^*, n2) + K1^* + n1^*, n3) + n1^*$$

The tag performs Tag Updating phase. The reader upon receiving message D performs the same calculation and compares D received with D^* calculated. If this is successful the reader performs Updating phase.

- Updating phase – the tag updates the two (IDS, K1, K2) tuples as follows:

$$n2^* = MIXBITS(n1^*, n3)$$

$$IDS_{old} = IDS$$

$$IDS_{next} = ROT(ROT(n1^* + K1^* + IDS + n2^*, n1^*) + K2^* \oplus n2^*, n3) \oplus n2^*$$

$$K1_{old} = K1$$

$$K1_{next} = ROT(ROT(n3 + K2^* + \pi + n2^*, n3) + K1^* + n2^*, n1^*) + n2^*$$

$$K2_{old} = K2$$

$$K2_{next} = ROT(ROT(IDS_{next} + K2^* + \pi + K1_{next}, IDS_{next}) + K1^* + K1_{next}, n2^*) + K1_{next}$$

The reader updates the back-end database using the following formulas:

$$n2^* = MIXBITS(n1^*, n3)$$

$$IDS = ROT(ROT(n1^* + K1^* + IDS + n2^*, n1^*) + K2^* \oplus n2^*, n3) \oplus n2^*$$

$$K1 = ROT(ROT(n3 + K2^* + \pi + n2^*, n3) + K1^* + n2^*, n1^*) + n2^*$$

$$K2 = ROT(ROT(IDS + K2^* + \pi + K1, IDS) + K1^* + K1, n2^*) + K1$$

The protocol requires exchanging four messages between the reader and the tag. All stages are illustrated in Figure 2.2. Hello message length is not specified, the IDS and D messages are 96-bits long and the concatenated A || B || C message consist of three 96-bit long sub-messages. A total of 384 bits (excluding Hello message) needs to be transmitted during one protocol run.

The Storage Requirements on the tag side are limited to 7 times the key-length (96-bits in the original specification) to hold two IDS, K1, K2 tuples and the static identifier ID. Each database record is required to store only one IDS, K1, K2 tuple and the static ID.

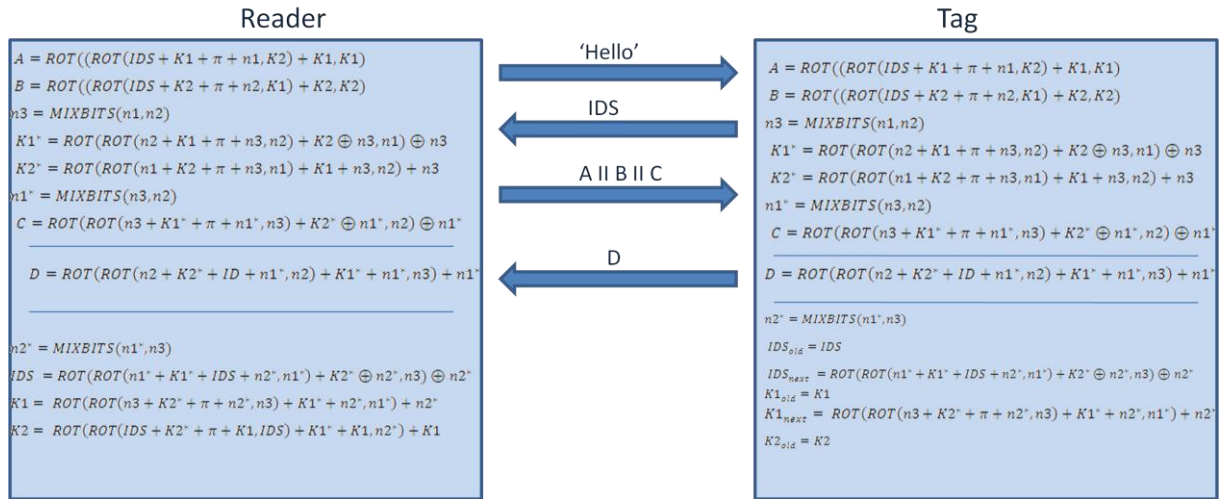


Figure 2.2 The Gossamer Protocol

The Gossamer Protocol prevents attacks listed in section 2.2.2 as follows:

- ID Disclosure Attack – the notion of an index-pseudonym (IDS) and private keys K1 and K2 changed for every authentication session prevents disclosure of the unique identifier (ID) of the tag.
- Full Disclosure Attack – the secret data (ID, K1, K2) is always scrambled using two random numbers and sum, Mixbits and Rot functions before being transmitted over the wireless link.
- De-Synchronization Attack – each tag stores (IDS, K1, K2) tuples used in a previous protocol run. In case of an unsuccessful update on the reader side in the last stage of the protocol (message D) the tag can be still identified using old values. The result is that both the tag and the reader can recover their synchronized state.

The requirement of the Data Freshness (see section 2.1) is fulfilled by updating secret values K1, K2, n1 and n2 at each protocol run.

To the knowledge of the author only one paper describing attacks against the Gossamer protocol was published (Ahmed et al. 2010) shortly before this dissertation was finished. Ahmed et al. described two attacks against the protocol. The first one was feasible if both random numbers n1 and n2 were equal to zero allowing the discovery of all secret values after eavesdropping two consecutive runs of the protocol. The latter attack concerned a case where both K1 and K2 values are equal to zero which leads to disclosure of all secret values during a single authentication round. Ahmed et al. proposed modifications to the protocol. However their proposal has a major flaw in that it renders the extraction of n1 and n2 impossible.

The original Gossamer protocol is given in fig. 2.1 replicated below for clarity of explanation.

$$A = ROT((ROT(IDS + K1 + \pi + n1, K2) + K1, K1))$$

$$B = ROT((ROT(IDS + K2 + \pi + n2, K1) + K2, K2))$$

An attack on the original Gossamer protocol is feasible if both random numbers $n1$ and $n2$ were equal to zero permitting the discovery of $K1$ and $K2$ after eavesdropping two consecutive runs of the protocol.

The proposed Ahmed et al modification substitutes $K1$ for $n2$ in A and $K2$ for $n1$ in B.

$$A = ROT((ROT(IDS + K1 + \pi + n1, K2) + K1, n2))$$

$$B = ROT((ROT(IDS + K2 + \pi + n2, K1) + K2, n1))$$

The values $n1$ and $n2$ are known to the reader. In A above, $ROT(IDS + K1 + \pi + n1, K2) + K1$ is rotated by $n2$ by the reader and likewise in B, $ROT(IDS + K2 + \pi + n2, K1) + K2$ is rotated by $n1$. Messages A & B are exchanged with the tag. The tag's job is to extract the values $n2$ and $n1$ from messages A and B and to perform the appropriate inverse rotation to verify the remainder of the contents of messages A and B. However, in this modification, the tag is not aware of the value $n1$ or $n2$ and therefore cannot perform the inverse rotation to retrieve $ROT(IDS + K1 + \pi + n1, K2) + K1$. This is a flaw in Ahmed's analysis that will not permit the completion of authentication.

Another modification proposed by Ahmed et al. concerning the MixBits function has also a very weak effect on overcoming the weakness of both random numbers $n1$ and $n2$ equal to zero. In the original Gossamer protocol, the mix-bits function exists during the creation of the new IDS and Key values.

```

Z = MixBits (X, Y)
Z = X
FOR counter = 0 to 32
Z = (Z>>1) + Z + Z + Y
ENDFOR
```

Figure 2.3 MixBits function (repeated)

Where X and Y are the input 96-bit numbers and Z is the final result of the MixBits function. The weakness identified by Ahmed et. al is that if both of the MixBits input values ($n1$ and $n2$ in the first run) are equal to 0 then the result of the function is also equal to 0. As a result all transformations are dependent on the Key values, the IDS and Pi. This weakens the effective security of the Gossamer Protocol. Ahmed et al proposed the following modification:

```

Z = MixBits (X, Y)
Z = X
FOR counter = 0 to 32
Z = (Z+counter) + Z + Z + Y
ENDFOR
```

Figure 2.4 Modified MixBits Function

It is obvious that in a case where both n_1 and n_2 numbers are equal to zero then the result of the MixBits function will be always the sum of numbers 1 to 32 which is 528. The proposed attack on the MixBits functions where n_1 and n_2 are 0 has been rectified but now the first attack proposed can be still performed but using the value of 528 instead of 0 at the first call of the MixBits function within the Protocol (n_3 calculation) and the result can be applied to the subsequent formulae to generate keys and messages.

Temporarily as a solution to the first attack it is recommended not to allow both random generated numbers to hold a value of zero at the same time. This verification should be performed by the PRNG function before the values are forwarded to the reader.

An altered Gossamer Protocol is suitable as a mechanism for authenticating Resource Limited Devices. The reader-tag relation is close to the master-slave one in the Infrastructure WSN scenario (see section 3.1). The main difference is the fact that a RFID tag is triggered by the reader, where in the IWSN all slaves will periodically initiate the communication. This difference is not significant in terms of the Gossamer specification as the 'Hello' message send by the reader to initiate the communication does not carry any protocol-specific data, thus can be discarded without any effect. It has to be pointed out that the Gossamer protocol was designed to be implemented in hardware but the simplicity of the mathematical operations renders it easily implementable in software on the reference platform nRF9E5 (see section 3.4). In consequence, this protocol is chosen for the implementation and further performance analysis.

Ultralightweight RFID Protocol with Mutual Authentication (UMA-RFID)

Shortly after the Gossamer Protocol was published Lee et al. proposed UMA-RFID alternative (Lee et al. 2009). The protocol is very similar to the Gossamer specification but simplified to use only bitwise operations (XOR, OR, AND) and a left bitwise rotation function ROT. Each tag contains a static identifier ID, pseudonym called the dynamic temporary identifier (IDT) and a secret key (K). All variables are 128-bits long and shared between the tag and the back-end database accessible by the reader (secure channel assumed). The reader is assumed to be capable of generating random number (N). The protocol consists of two stages: Authentication Phase and Update Phase.

- Authentication Phase: the reader sends a request message and the tag replies with a temporary identifier (IDT). The reader searches the database to find a secret key K_i corresponding to the IDT received, generates random number N_i and calculates messages A_i and B_i as follows:

$$A_i = K_i \oplus N_i$$

$$B_i = ROT(K_i, K_i) \oplus ROT(N_i, N_i)$$

The messages are concatenated and sent to the tag. Upon receiving these messages the tag obtains N_i from message A_i and calculates message B_i' in the same way as the reader previously. Then message B_i' is compared with B_i . If they are the same then the reader is authenticated and the tag generates reply message C_i as follows:

$$C_i = (K_i \vee ROT(N_i, N_i)) \oplus (ROT(K_i, K_i) \wedge N_i)$$

The message is sent to the reader and the reader calculates a local copy and verifies the correctness. After successful verification the tag is authenticated.

- Updating phase: the tag performs this phase after authenticating the reader. The updating on the reader side is done upon successful authentication of the tag. Both sides use the following equations to update IDT_{i+1} and K_i :

$$IDT_{i+1} = K_i \oplus ROT(N_i, N_i)$$

$$K_i = ROT(K_i, K_i) \oplus N_i$$

Peris-Lopez et al. analysed the UMA-RFID protocol (Peris-Lopez et al. 2009) and found serious weaknesses in the scheme which led to ID Disclosure, Full Disclosure and De-Synchronization attacks. Peris-Lopez et al. described 5 attacks: ID-disclosure attack, two passive Full Disclosure attacks and two active De-Synchronization attacks. The most significant Full Disclosure attack allowing cloning of the tag to be performed after eavesdropping of only two consecutive runs of the protocol and requires only computing XOR among some of the messages transmitted over the radio channel.

SQUASH – A New MAC with Provable Security Properties for Highly Constrained Devices Such as RFID Tags

Adi Shamir proposed an authentication mechanism based on a challenge-response scheme and Message Authentication Code (MAC) called SQUASH (short for SQUare-hASH) specifically for Resource Limited Devices (Shamir 2008). The proposed challenge-response scheme allows tag-to-reader authentication and does not address the ID disclosure issue. The document focuses on describing a strong one-way hashing function (H) performed by the tag upon receiving a random challenge message (R). The MAC is computed with (R) and secret key (S) as inputs:

$$MAC = H(S, R)$$

The reader shares the secret key S and performs the same calculation upon receiving the MAC to validate if a tag is legitimate. The author made an interesting observation that most of the standard one-way hash functions such as SHA-1 (Eastlake & Jones 2001) are primarily designed to be collision resistant as their main area of usage concerns digital signatures. The requirement for collision resistance typically adds complexity to the algorithm. Since a collision is not a security threat in

a challenge-response scheme, the author proposed an algorithm based on the Rabin encryption scheme (Rabin 1979). In the Rabin scheme the ciphertext (c) is computed as $c = m^2 \pmod{n}$, where (m) is a message and (n) is a product of at least two unknown prime factors. Shamir has shown how the calculation can be simplified using a step-by-step process that has no adverse effects on the strength of the security and has proposed a hardware implementation using mixing function (M) applied to the secret and challenge (S, R) and then the SQUASH function $SQUASH(M(S,R))$ as follows:

1. Start with j which is the index at lower end of the desired extended window of $t + u$ bits, and set carry to 0.
2. Numerically add to the current carry (over the integers, not modulo 2) the k products of the form $mv * m_{j-v \pmod{k}}$ for $v = 0, 1, 2, \dots, k - 1$.
3. Define bit c_j as the least significant bit of the carry, set the new carry to the current carry right-shifted by one bit position, and increment j by one.
4. Repeat steps 2 and 3 $t + u$ times, throw away the first u bits, and provide the last t bits as the response to the challenge. (Shamir 2008)

The proposed SQUASH-128 hash function uses a modulus $2^{1277} - 1$, a 64-bit key S and a 64-bit challenge R to produce a 32-bit response. The security of this scheme was questioned by Ouafi & Vaudenay, who discovered a key recovery attack known as "known random coins attack" against the Rabin scheme using 1024 chosen challenges (Ouafi & Vaudenay 2009). The "known random coins attack" allows an adversary to request many encryptions of the same plaintext and in consequence get the random coins. The attack is only effective if a linear mixing function is used, thus the security of SQUASH is still regarded as strong, assuming that a non-linear mixing function is used.

SPINS - Security Protocols for Sensor Networks

Perrig et al. proposed a security mechanism consisting of two blocks: Secure Network Encryption Protocol (SNEP) and μ TESLA (Perrig et al. 2002). SNEP's security goals are data confidentiality, mutual authentication and the evidence of data freshness². μ TESLA provides a mechanism for an authenticated broadcast.

According to the authors SNEP achieves previously mentioned security goals with a very low communication overhead of only 8 bytes per message. SNEP ensures semantic security³ using two counters C_A and C_B shared by the communicating nodes. These counters are further used by the block cipher in counter mode. Counters do not have to be attached to messages but Perrig et al. described a mechanism of counter synchronization. The mutual authentication is achieved through the usage of a MAC function. Both communicating nodes A (sender) and B (receiver) share a master secret key X_{AB} used to derive keys through a pseudorandom function. It has to be noted that the authors advised deriving different

² Data freshness ensures that the data received is fresh and the adversary cannot replay old messages.

³ Semantic security ensures that an eavesdropper is not able to deduce any information about the plaintext even after analysis of multiple encryptions of the same plaintext.

key sets for MAC and encryption. Each key set consists of two keys - one for each direction of the communication.

Encryption keys:

$$K_{AB} = F(X_{AB})$$

$$K_{BA} = F(X_{AB})$$

MAC keys:

$$K'_{AB} = F(X_{AB})$$

$$K'_{BA} = F(X_{AB})$$

The sender node A encrypts (symmetric block cipher) the data using K_{AB} and C_A :

$$E = F_{crypt}(K_{AB}, C_A)$$

The encrypted result is then used by the MAC function in the following manner:

$$MAC(K_{AB} C_A \parallel E)$$

Finally the sender node sends the message:

$$A \rightarrow B: F_{crypt}(K_{AB}, C_A), MAC(K'_{AB} C_A \parallel F_{crypt}(K_{AB}, C_A))$$

This scheme does not provide data freshness. A solution for this issue is provided by the usage of a random number N_A and a request message R_A send by the node A:

$$A \rightarrow B: N_A, R_A$$

The Receiving Node B responds with an encrypted message and a MAC function with a cryptographic nonce N_A as one of the inputs:

$$B \rightarrow A: F_{crypt}(K_{BA}, C_A), MAC(K'_{BA}, N_A \parallel C_B \parallel F_{crypt}(K_{BA}, C_B))$$

Upon receiving the message and MAC verification the node A is sure that the node B generated the message using the cryptographic nonce supplied in a request message.

SNEP messages require synchronized counters on both sides of the communication. If the synchronization is lost for example due to lost messages, counter values can be re-synchronized through the following messages:

$$A \rightarrow B: C_A$$

$$B \rightarrow A: C_B, MAC(K'_{BA} C_A \parallel C_B)$$

$$A \rightarrow B: MAC(K'_{AB}, C_A \parallel C_B)$$

The concept of μ TESLA was based on a TESLA protocol providing a mechanism of an authenticated broadcast (Perrig et al. 2001). This scheme achieves asymmetry through a delayed disclosure of symmetric keys rather than using computationally expensive Public Key Cryptography. The TESLA proposal is not suitable for implementation within a constrained devices environment. In order to adapt it for the Wireless Sensor Networks the following issues were addressed:

- TESLA authenticates the initial packet with a digital signature. The computation of a digital signature is too expensive on sensor nodes so μ TESLA uses only a symmetric mechanism.
- Standard TESLA discloses the key for the previous intervals with every packet. Since this generates too much overhead μ TESLA discloses the key once for each pre-defined epoch.
- Sensor nodes are not able to store an entire one-way key chain in the memory. This is addressed in μ TESLA by limiting the number of authenticated senders.

The μ TESLA requires that all receiving nodes are loosely time synchronized with the base station. In order to send an authenticated broadcast, the base station computes a MAC using the packet and a key which is secret at that point in time. The receiving node stores the packet in the buffer in order to validate its authenticity later when the base station broadcasts the verification key. Each MAC is a key of a key chain, generated by applying a one-way hash function. A successive key is generated by applying the hash function on the previous key. The time synchronization can be achieved by the means of the SNEP protocol. The protocol consists of the following phases:

- Sender setup – the sender node generates a one-way key chain by successively applying one-way hash function.
- Broadcasting authenticated packets – the time is divided into inform intervals. The sender associates each key in the key chain with one particular interval and uses this key to compute MAC of all packets sent in that interval. The key K_i is disclosed after a delay which is greater than a few time intervals and has to be greater than a message round-trip time between the sender and the receivers.
- Bootstrapping a new receiver – each receiver needs to authenticate one key in the one-way key chain which allows it to commit to the entire chain: further keys will be calculated using one-way hash function. The receiver needs to be loosely time synchronized with the sender and has to know the key disclosure schedule. Both of these requirements are fulfilled as follows: the Receiver node sends a request message containing a random number N_R and the Sender replies with a message containing its current time T_S , the key K_i used in the past interval i , the starting time of this interval T_i , the duration of this interval T_{int} and the key disclosure delay δ . These values are sent with clear

text along with a MAC calculated using these values, the random number N_R and shared secret key K_{MS} :

$$B \rightarrow A: N_R$$

$$A \rightarrow B: T_S \parallel K_i \parallel T_i \parallel T_{int} \parallel \delta$$

$$A \rightarrow B: MAC(K_{MS}, N_R \parallel T_S \parallel K_i \parallel T_i \parallel T_{int} \parallel \delta)$$

- Authenticating broadcast packets – upon receiving a message the receiver needs to make sure that this packet is safe by verifying if the key used to compute the MAC was not disclosed yet. This can be achieved thanks to loose time synchronization between the sender and the receiver. When a node receives a new key K_i it computes the one-way hash function on the previous key K_v in order to verify the correctness of K_i . If the check was successful then a node can authenticate all packets which arrived in the time interval of K_i .

Perrig et al. did not specify exactly which encryption algorithm should be used in SNEP, or which one-way hash function should be used by μ TESLA, or indeed which Random-number generation should be used in both protocol blocks. However, Perrig et al. provide example functions for the experimental implementation. In order to tackle the issue of limited code space and RAM size all cryptographic primitives are based on a modified subset of the RC5 encryption algorithm (Rivest 1995).

The μ TESLA's main disadvantage is the need for an initial unicast-based parameter distribution. This issue has been addressed by Liu & Ning in the Multi-Level μ TESLA specification (Liu & Ning 2004). The scheme provides a way to predetermine and broadcast the initial parameters. Additionally, Multi-Level μ TESLA introduces a multi-level key chain scheme which removes the need for very long key chain. The authors claim that the key chain commitment distribution mechanism described in their document improves the survivability of the scheme against message loss and Denial Of Service (DOS) attacks.

Since the SPINS specification does not propose exact cryptographic primitives to be used no security weaknesses were identified in the scheme to the knowledge of the author. However, several papers were published addressing efficiency and key management issues found in SNIPS (Liu & Ning 2004), (Yu-Long et al. 2007), (Hegazy et al. 2007).

Both of the SPINS schemes suffer from using Pseudo-Random Number Generator (PRNG) engines not only on the base station side but also on the sensors. Perrig et al. suggested that sensor nodes may draw random numbers from the actual sensor readings. However, the Analog-to-Digital Converters (ADCs) which are sometimes only 8 or 10-bit wide may not be able to provide random values in a magnitude large enough for cryptographic usage. Thus a resource expensive PRNG functions need to be implemented within a limited sensor node code space.

Due to a high computational overhead on the sensor node side, the SPINS implementation was not considered during this MSc project.

2.3 Encryption

Bruce Schneier said that “Cryptography is the art and science of keeping messages secure”(Schneier 1996). A message (referred to as a plaintext) undergoes a process of hiding its substance (encryption) and converting it into a non-meaningful gibberish (ciphertext or cipher) that can be sent over an insecure communication channel. The process of retrieving a plaintext from a ciphertext received is referred to as decryption.

The general rule followed in modern cryptography states that the security of the system cannot rely on the secrecy of its components (security by obscurity) – the secrecy must reside entirely in the encryption key. This principle was stated by Auguste Kerchoff in the nineteenth century (Menezes et al. 1997), who assumed that a cryptanalyst has a complete knowledge of the algorithm and implementation. An algorithm that has its security based on keeping its foundations secret is called restricted. Such a security system can be compromised through an information leak, reverse engineering, etc. Quality control and standardisation cannot be maintained.

The most common type of cryptography is the Secret-Key Cryptography (symmetric cryptography), where a message ‘M’ gets encrypted with encryption function E, using a key ‘k’ to generate a ciphertext ‘c’. Therefore, $c = E(k,M)$. The decryption function D should provide a way to recover the plaintext ‘p’ using shared secret ‘k’, such that $p = D(k,c)$.

2.3.1 Problem of Encryption in the context of IWSN

Resource-Limited Devices (RLDs) are highly constrained in terms of available memory and processing power (see Section 3.3). The reference platform nRF9E5 (see Section 3.4) used in the example IWSN does not provide any hardware support for any encryption algorithm, thus the entire mechanism needs to be implemented in software. The following characteristics are used to evaluate possible encryption algorithms:

- The code-space required for the implementation is limited (algorithm simplicity) and will be used as one of the metrics.
- The algorithm should be optimized for 8-bit word size.
- The expected data payload size is limited, thus the resource efficiency of the algorithm will take precedence before the data throughput.
- The single data payload size is limited to 24 bytes on most occasions.
- It may be not possible to implement a random number generator on the node side due to general hardware and execution time constraints.
- Thanks to the proposed authentication algorithm (see Section 2.2.3) the key management problem may be resolved through a re-use of the authentication

key for the purpose of a session key fed into a symmetrical block or stream cipher.

The most security critical aspect of wireless sensor operation is the reconfiguration of the nodes. An attack enabling an adversary to alter the control messages may lead for example to Denial Of Service (DOS) attacks affecting the entire network of sensors. It is assumed that the authentication system will guarantee frequent session key changes for the purpose of maintaining the data freshness (see Section 2.1). In consequence, the control messages have to remain safe for a relatively short period of time, until the next session key is exchanged. In most IWSN applications the data transferred by sensors will not be valuable to an attacker and will not require infinitely long secrecy. Given the analysis above, the encryption algorithm may be based on relatively short encryption keys.

2.3.2 Known and possible attacks

Attacks can be generally divided into two categories - passive and active attacks. Passive attacks concern monitoring the communication channel and gathering data (eavesdropping) but not altering it in any way. Wireless Sensor Networks are especially prone to passive eavesdropping.

Cryptanalysis is an area of science heavily used in performing passive attacks on Encryption Algorithms. It concerns recovering plaintext of a message without knowing the Encryption Key. Schneier divided cryptanalytic attacks into several groups (Schneier 1996). The most important are as follows:

- Ciphertext-only attack – the attacker analyses the ciphertext of several messages encrypted with the same algorithm in order to recover the encryption key.
- Known-plaintext attack – the attacker has access to a block of plain text and a ciphertext produced by the algorithm out of this block. The analysis tries to extract information on the encryption key by examining changes between input and output.
- Chosen-plaintext attack – similar to the known-plaintext attack but the attacker is able to choose a plaintext block to be encrypted.
- Adaptive chosen-plaintext attack – the cryptanalyst sequentially applies chosen-plaintext attack on variable size plaintext blocks, where each choice is dependent on the outcome of previous attack.
- Chosen-ciphertext attack – this kind of attack assumes that the cryptanalyst has access to the decrypted plaintext corresponding to the ciphertext he chose (without knowing the decryption key).

Attacks requiring alteration of the transmitted ciphertext or alteration of the computation in a device are referred to as active. These types of attacks commonly target specific protocol implementation of the security system rather than cryptographic algorithms on their own.

A special case of an active attack is a physical invasive attack - the adversary has a physical access and toolset required to access the device's circuitry and for example read the EEPROM memory contents. Since a full protection against this types of attack requires advanced hardware (such as a sensor case destroying the EEPROM chip during opening), it is assumed that such attack cannot be prevented. The consequences of the physical invasive attack have to be limited in such a way that the security of the entire system is not compromised when a single node's key is revealed and the past communication remains safe (see Section 2.1). This issue introduces a requirement to maintain session keys unique to each of the sensor nodes.

2.3.3 Identified algorithms effective in the context of Infrastructure WSN

The field of research concerning cryptography for low cost embedded devices was not given much attention until the last two decades. The papers concerning lightweight cryptography are focused either on finding solutions easily implementable in hardware (Bogdanov et al. 2007, Eisenbarth et al. 2007, Poschmann et al. 2007) or solutions focused on a software implementation efficiency on low resource microcontrollers (Standaert et al. 2006, Wheeler & Needham 1994). The reference platform used in this research forces a software encryption solution.

Tiny Encryption Algorithm (TEA) family

The Tiny Encryption Algorithm (TEA) (Wheeler & Needham 1994) was the initial proposal of a family of algorithms (chronologically): XTEA and Block TEA (Needham & Wheeler 1997) and XXTEA (Wheeler & Needham 1998). The main principle behind the TEA algorithm design was the simplicity of the implementation and the ease of translation to many programming languages (including Assembly). The initial proposal was a block cipher operating on 64-bit blocks with 128-bit key. Each of the identical 64 rounds of the algorithm uses only logical AND, OR, as well as bit-shift operations and addition/subtraction Mod 2^8 . The sample C-language source code consisted of less than 10 lines. The authors favoured large number of iterations over the complexity of the code. The set up time is relatively short and there is no need to store any Look-Up-Tables (LUTs) in the memory.

The first weakness discovered in TEA was the fact that each key is equivalent to three others which effectively reduces the key size to 126 bits. This vulnerability was used to construct an attack against Microsoft's Xbox game console, which uses TEA as a hash function (Russell 2004). Since the initial proposal in 1994 several attacks were published, for example a Key-schedule cryptanalysis (Kelsey et al. 1997) and Related-key cryptanalysis (Kelsey et al. 1997). Wheeler & Needham addressed the issue mentioned above when proposing Block TEA and XTEA algorithms. The key schedule was revised and other computations (bit-shifts, XORs and additions) were rearranged to introduce the key material more slowly. The XTEA algorithm and its block version Block TEA also suffer from weaknesses discovered by shortly after publication by Saarinen (Saarinen 1998): slow diffusion in the decryption direction

exploited by chosen plaintext attack. Several other cryptanalysis attempts were also published in (Andem 2003, Hong et al. 2004, Ko et al. 2004, Lu 2009, Moon et al. 2002). The slow decoding propagation pointed by Saarinen was addressed by Wheeler & Needham in their XXTEA proposal as a short amendment to the Block TEA (Wheeler & Needham 1998).

XXTEA operates on a block consisting of at least two 32-bit words using a 128-bit key. A single round of the algorithm can be viewed as operations on a word and its two adjacent words (previous and the next one). Figure 2.5 shows one round of XXTEA cipher, where x_r represents a current block and the four-squares symbol represents addition Modulo the size of the word. The number of rounds equivalents to the number of words in the block.

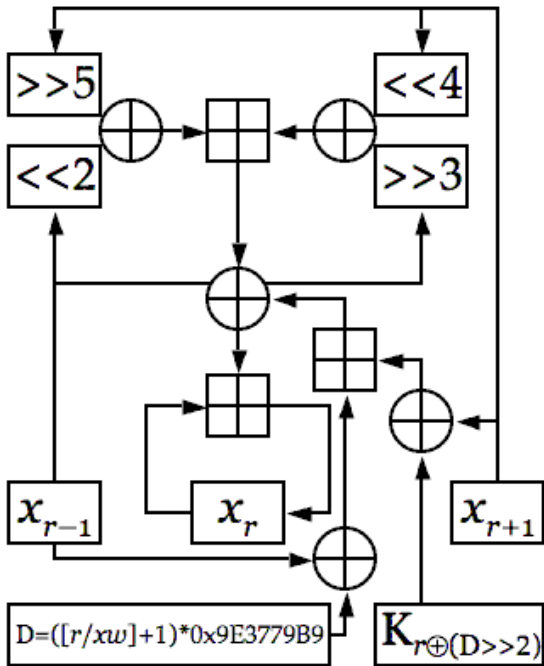


Figure 2.5 One round of XXTEA (el Ruptor 2007)

In (Rinne et al. 2007), Rinne analysed the performance of several ciphers, including DES (Federal Information Processing Standards 1993), AES (Daemen & Rijmen 1999), IDEA, SEA (Standaert et al. 2006), HIGHT and the TEA family. Rinne indicated that the TEA family requirements in terms of the code space required are among the lowest (after the IDEA algorithm) throughout all ciphers analyzed. The small code space footprint was achieved thanks to the lack of substitution tables common in other block ciphers. The XXTEA optimisation and performance analysis were also provided in (Jinwala et al. 2008) proving it to be a viable encryption algorithm for WSNs.

Shortly before this dissertation was completed E. Yarrkov published a chosen-plaintext attack against the XXTEA requiring 2^{59} queries (Yarrkov 2010). Yarrkov took advantage of the fact that the number of full cycles to perform over each block is equivalent to $6 + 52/n$, where n represents the number of rounds. If the block

consists of at least 53 words then the number of cycles per word is reduced to only 6. This characteristic was used to perform differential cryptanalysis, where the difference was considered subtraction per word. The author described two attacks proving that XXTEA does not provide the intended 128-bit security.

Scalable Encryption Algorithm (SEA)

The Scalable Encryption Algorithm (SEA) proposed in (Standaert et al. 2006) aims to provide a low cost encryption implementable on resource limited processors. Similarly to the TEA family it uses basic operations such as logical AND, OR, XOR, word/bit rotations, modular additions and a simple substitution box. Apart from the limited instruction set, the other design criteria were the low memory requirements and small code size. Per the authors the algorithm allows “on-the-fly” key derivation. The proposal includes a comprehensive security analysis showing the resistance of the protocol against major modern cryptanalytic techniques.

The scalability of the algorithm is achieved through the flexibility in the size of the input parameters. The following parameters are used:

n: plaintext size, key size

b: processor (or word) size

$n_b = \frac{n}{2b}$: number of words per Feistel branch

n_r : number of block cipher rounds.

There is one constraint on the size of the key/plaintext that the *n* is a multiple of 6*b*:

$$n = x6b$$

The security analysis provided in the proposal suggests the minimum required number of rounds to provide security against well know attacks (assuming word size equal or greater than 8 bits) is:

$$\frac{3n}{4} + 2(n_b + \frac{b}{2})$$

Figure 2.6 shows one encryption and key round, where *R* denotes the word rotation, *r* the bit rotation and *S* the substitution box ($S_T := \{0,5,6,7,4,3,1,2\}$ in C-like notation). The $C(i)$ represents a n_b -word vector with all words of a value 0 except the least significant word which value is equal to *i*. The *Li* and *Ri* represent left and right halves of the word or the key (*KLi*, *RKi*).

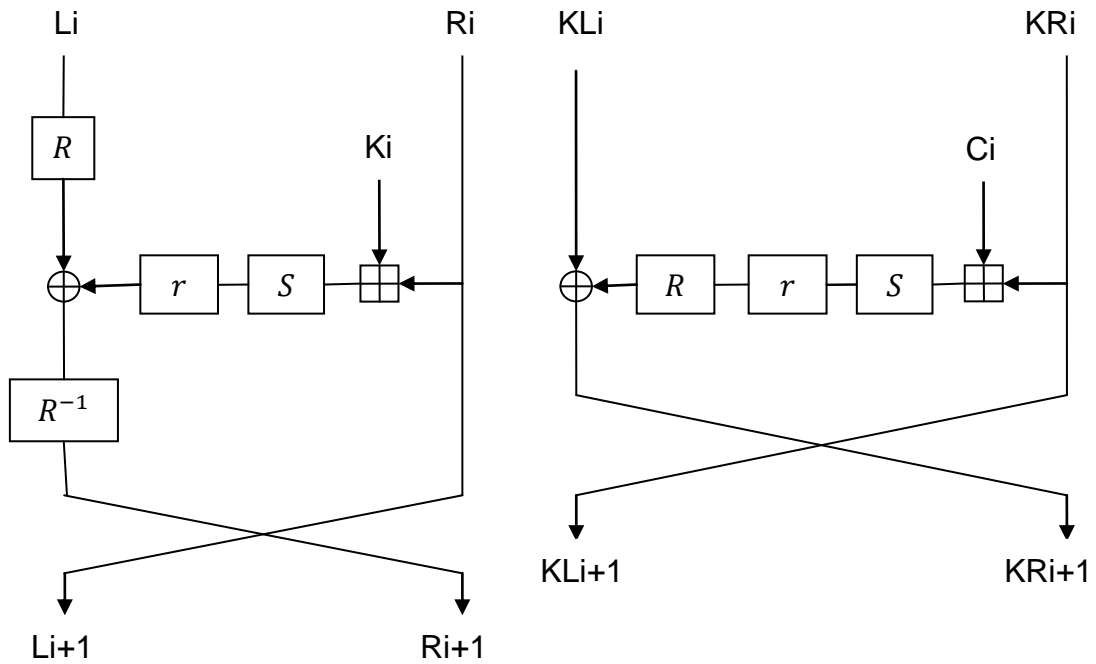


Figure 2.6 Encrypt/decrypt and key round of SEA

The functions for encrypt (F_E), decrypt (F_D) and key (F_K) rounds are defined as follows:

$$F_E(L_i, R_i, K_i): R_{i+1} = R(L_i) \oplus r(S(R_i \boxplus K_i)), \quad L_{i+1} = R_i$$

$$F_D(L_i, R_i, K_i): R_{i+1} = R^{-1}(L_i \oplus r(S(R_i \boxplus K_i))), \quad L_{i+1} = R_i$$

$$F_K(KL_i, KR_i, C_i): R_{i+1} = R^{-1}(KL_i \oplus R(r(S(KR_i \boxplus C_i)))), \quad KL_{i+1} = KR_i$$

The authors analyzed the performance of the algorithm using the Atmel AVR ATiny reference 8-bit CPU platform among others. The expected code size for a 96-bit key implementation was estimated at 386 bytes and the amount of clock cycles required for encryption/decryption was estimated at 17745. A performance analysis (Rinne et al. 2007) performed by Rinne on AVR Atmel163 showed a code size of 2132 bytes for the 96-bit SEA (compared to 1160 bytes for XTEA) and the number of CPU cycles required to complete encryption/decryption was 9654 (compared to 6718 with XTEA).

The performance and code space requirements of the XTEA algorithm look more promising than the SEA according to Rinne's analysis. However, due to the recent Yarrkov's discovery of security weaknesses in XXTEA the implementation of this algorithm will be abandoned in favour of the Scalable Encryption Algorithm in 96-bit version.

3. Resource-Limited Devices

The term Resource-Limited Device (RLD) will be further used to describe a microcontroller device with significantly lower processing power and limited memory in comparison to a modern Personal Computer. This group of devices range from Radio Frequency Identification (RFID) transponders to a wide spectrum of embedded devices equipped with small (typically 8-bit) microcontrollers. Such devices are utilised in wireless sensor networking for example.

This dissertation focuses on the security of the communication over the radio channel, thus the area of research will be restricted to Wireless Sensor Networks and advanced RFID systems.

The work in this MSc dissertation is predicated upon the application of the Nordic Semiconductors nRF9E5 Integrated Circuit (Nordic Semiconductors 2009b) as the target device. This microcontroller was chosen due to its low price (approximately 2\$US per unit at quantities over 1000), integrated UHF radio transceiver and excellent power saving characteristics which make it an ideal solution for the design of a low-cost wireless sensor. The nRF9E5 entire chip will be referred to as a microcontroller and the Intel 8051-compatible Central Processor Unit - a subset of this system will be referred to as CPU or microprocessor.

Hardware Platform - nRF9E5

The nRF9E5 microcontroller is a single chip system with an integrated sub-1GHz Radio-Frequency (RF) transceiver, 8-bit 8051-compatible processor and 4-input 10-bit Analogue to Digital (AD) converter. The design of the chip was based on the Dallas DS80C320 CPU in terms of hardware specification and instruction cycle timing. It is a low cost solution with extended power saving capabilities. The minimum power consumption in power down mode (where the chip can be woken up by a timer or an external pin) is only 2.5 μ A. The microprocessor draws 2.2 mA of current at a clock frequency of 16MHz and the radio transceiver (nRF905) uses 10 to 30 mA in Transmit Mode (depending on output power setting). Receive Mode power usage is estimated at 12.5 mA on average.

The CPU is an 8-bit Intel 8051 derivative with the addition of Special Function Registers (SFRs) used to control the nRF9E5 radio transceiver. The microcontroller is equipped with 512 bytes of ROM that contains the bootstrap loader, 256 bytes of Internal Data Memory, 128 SFRs and 4 kilobytes of external on-chip RAM. The memory is organized with the Harvard Architecture in contrast to the Von Neuman architecture commonly used in desktop PCs. The bootstrap loader loads the program from the bottom area of external EEPROM memory upon each power-up or reset cycle. The manufacturer did not provide any possibility to extend the size of the on-chip RAM, thus the binary program size is limited to 4 kilobytes.

The on-chip radio transceiver subsystem is the Nordic Semiconductors nRF905 connected to the microcontroller through the SPI (Serial Peripheral Interface) port. It utilizes the nRF ShockBurst™ technology allowing high speed radio signal processing without the assistance of the microprocessor, which further reduces the power consumption requirements. The transceiver is able to generate the preamble and calculate CRC for each data payload when transmitting signals. Additionally, it can validate CRC for each incoming data payload. The CRC calculations are performed by an on-board circuit without the CPU's assistance. NRF905 supports standby mode, where the current consumption is limited but the short startup times are still maintained, and 4 different Radio Frequency (RF) transmitting power modes ranging from -10dBm (at 9mA of current consumption) to 10dBm (at 30mA of current consumption). The current consumption in receiving mode is estimated at 12.5mA and can be reduced to 10.5mA when using reduced receiving power mode. The modulation used for the air interface is Gaussian Frequency Shift Keying (GFSK) with Manchester Encoding yielding an effective data transfer rate of 50kbps. The transceiver is able to operate on radio frequencies 430 to 434.7MHz or 868 to 928MHz.

3.1 IWSN introduction.

Classic WSN

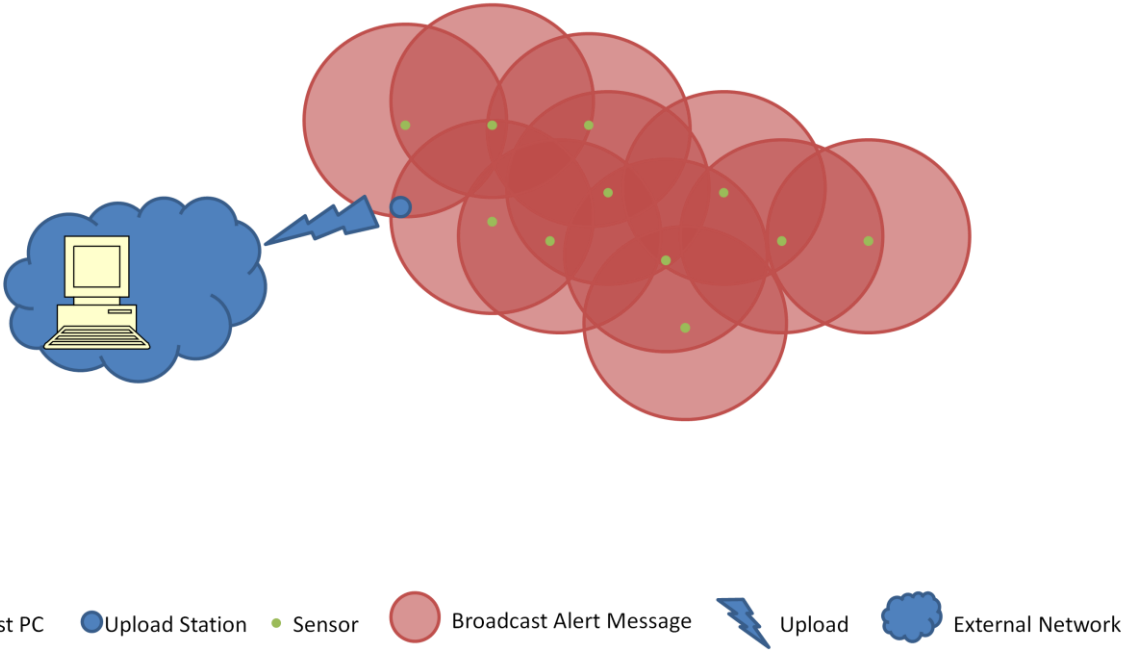


Figure 3.1 Wireless Sensor Network Architecture

A typical Wireless Sensor Network consists of a set of Wireless Sensor Nodes and one or more Upload Stations (also referred to as Gateway Sensor Nodes or sinks) (Akyildiz et al. 2002) that provide a connection to a Host Computer on an external network. The external network uses a communication media not available to the Wireless Sensor Nodes, such as Ethernet or a different RF technology. The most commonly used architecture (Ye et al. 2002) is an ad-hoc network (see Figure 3.1), where every sensor node either broadcasts the message to all other nodes (using an endless message repetition preventing mechanism) or uses a routing mechanism to forward the message to the upload station through a series of other sensor nodes used as 'hops'(Kamble et al. 2007). Once the upload station receives the message it is uploaded to the External Network.

Such architecture is useful in applications where sensors are distributed in an unplanned manner (e.g. battlefield sensor network deployed from an aircraft) and messages can be sent unreliably with no confirmation of the delivery from the Upload Station (although the acknowledgement system can be implemented in this architecture if the routing mechanism allows that).

Infrastructure Wireless Sensor Network (IWSN) example (HINT Project 2010) used further in this dissertation describes an architecture consisting of the following:

- Master device - an equivalent of the Upload Station in Classic WSN, linked to the External Network using for example an Ethernet controller or 802.11 WiFi controller.
- Sensor (Slave) - a battery operated Wireless Sensor Node in the network equipped with microcontroller, radio transceiver and ADC converter allowing readings from the attached sensors.
- Repeater - a bridge forwarding messages between wireless sensors and a master device. Uses similar radio hardware to sensors but is assumed to have a regulated power source.
- Host PC - Host computer used by an operator to control the IWSN.

This type of architecture (see Figure 3.2) can be found in WSN with a planned distribution of sensors, e.g. a network of temperature monitoring sensors deployed within a large building. It is assumed that all devices operate on the same radio frequency. All battery operated sensors (Slaves) attempt to connect to the Master device at a pre-programmed interval. The Master device uses an acknowledgement mechanism to guarantee the delivery of a single packet or an entire multi-packet transmission (depending on the configuration and packet type). Each Slave repeats the transmission attempt a pre-programmed number of times if an acknowledgement was not received. Repeaters are used to extend the coverage area of the network by forwarding each received packet to the Master (or other Repeater) and forwarding acknowledgements back to Sensors.

Infrastructure WSN

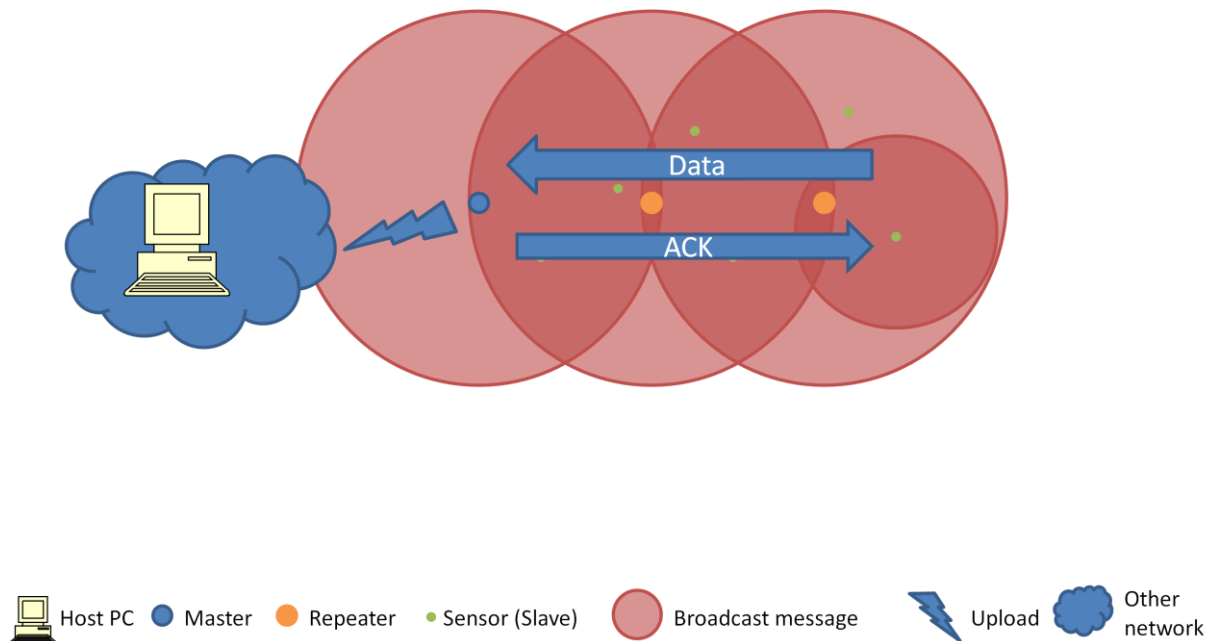


Figure 3.2 Infrastructure Wireless Sensor Network Architecture

All devices using the radio link utilise a simple collision avoidance mechanism with a back-off system similar to Pure-Aloha Protocol (Abramson 1970): every node listens to determine if the radio carrier is busy prior to a transmission attempt. If the carrier is sensed as busy then a node backs off for a random period of time before another transmission attempt.

The main advantage of this architecture is the simplicity of communication between devices as no routing tables need to be maintained, even though the delivery of specific (user pre-defined) data packets can be confirmed by the acknowledgement mechanism. Thanks to this simplicity the radio-handling part of the software can be implemented within the limited code space and run efficiently on many Resource Limited Devices. However, this architecture is not ideal in environments, where Repeaters and Masters cannot be provided with a fixed power source. The other disadvantage is that as more slaves are introduced to the network performance degrades. The simplicity of the collision avoidance mechanism inhibits the use of a large number of slaves because slaves share a common frequency channel for transmission.

3.2 Description of the technical problem of authentication and encryption in the context of the IWSN.

Infrastructure WSNs experience common issues related to the use of modulated radio frequency spectrum (radio waves) as the communication medium: Eavesdropping is possible on any wireless link using virtually any radio transceiver tuned to a given frequency with the ability to demodulate the signal. The architecture (described in 3.1) assumes that the link between the Master and the Host is secure. The Repeaters are used only to receive and forward data packets, without processing them and act as radio range extenders. Repeaters will not perform any active role in the security mechanism. The only parties requiring mutual authentication and secured (encrypted) communication channel are the Master and the Slave.

Since the encryption and decryption mechanism has to be implemented on the Slave device, choosing such a mechanism must involve consideration of the limitations listed in Section 3.3. Ideally, the Master device will have an always-on, secured link with a Host (server) and this Host device can perform all of the computationally heavy encryption and decryption-related calculations. In other words, the master can offload all computationally heavy tasks to a back-end server and accept returned values. This permits the processing power of the master device to be used to handle service requests from a number of slave devices, rather than becoming occupied with computations associated with authentication and encryption.

3.3 What are the specific problems associated with Resource Limited Devices

The constraints imposed on possible implementations of security systems for RLDs can be categorised as Central Processor Unit (CPU) limitations, memory limitations, power consumption and cost barriers.

CPU constraints

The main CPU constraint in resource-limited devices is obviously the limited processing power of the processor. Passive RFID transponders (powered by an external interrogator) with a very limited number of logic gates on the circuit may be only capable of performing simple logical operations with one-bit values. More powerful embedded devices may be using 8-bit CPUs for example the Intel 8051 derivative nRF9E5 clocked at 12MHz, which is able to execute only 750,000 operations per second (assuming that 50% of operations require two CPU cycles and the remaining require one).

The number of operations per second is not the only constraint relating to the processor. Another issue related to microcontrollers is the word size. The most

commonly used cryptographic standards were designed to be implemented either in hardware (e.g. the first proposals of the Data Encryption Standard - DES (Federal Information Processing Standards 1993)) or more flexible using software. However, the majority of standards assume that 32-bit CPUs will be used, thus their mathematical basis and implementation is commonly optimized to use 32-bit (e.g. the Rijndael cipher (Daemen & Rijmen 1999)) or even 64-bit word. 8-bit microcontrollers would be forced to perform numerous instructions to handle 32-bit numbers manipulation, e.g. it takes approximately 35 CPU operations to multiply two 32-bit numbers on the 8051 8-bit CPU (Vault Information Services 2009).

Memory limitations

Low-cost passive Electronic Product Code (EPC) RFID tags can have as little as 104 bits of non-volatile memory (EPCGlobal 2008) and may not even contain any Random Access Memory. More advanced tags however, may be equipped with 1-2KB of memory. Microcontrollers are typically equipped with no more than 64KB memory, but this amount can be subject to limitations also due to 8-bit addressing issues causing slow access to some parts of the memory.

Heavyweight cryptographic techniques using large keys (even 2048-bit in some RSA implementations) cannot be implemented in resource-limited device environments not only due to the amount of memory needed but also due to slow memory access times and limited read/write lifecycle. Most of the EEPROM memory chips allow for one million Read/Write cycles.

Power consumption

Resource-limited devices are heavily constrained in terms of power availability for their operation. Passive RFID tags draw the whole power from the interrogating device; active RFID solutions and wireless sensors are often powered by small cell batteries and are expected to provide a reasonably long operation time between battery replacements. Wireless Sensors are typically designed for one-time use, thus their lifetime can be increased only by power saving. All CPU-intensive operations and memory manipulations required by most cryptographic algorithms along with the radio transceiver usage are the most power consuming activities performed by such devices so they have to be limited to a minimum.

Cost Barriers

Most of the resource-limited devices are designed to be manufactured in high volumes with a very low price per item. An addition of a single logic gate to the electronic circuit may seem inexpensive but if multiplied by millions of manufactured items may have a substantial influence on the profit made by the manufacturer. This constraint forces solutions requiring little or no additional hardware modifications.

3.4 Technical description of the processor and its implications for effective security implementation

The nRF9E5 single chip system uses an 8-bit microprocessor with an instruction set compatible with the industry standard 8051 processor. The instruction timing differs from the industry standard: each instruction uses 4 to 20 clock cycles instead of 12 to 48 in the standard. The hardware specification of the chip (Nordic Semiconductors 2009b) allows utilizing a 4-20MHz crystal oscillator to generate clocking signal on the circuit (shared by microcontroller, AD converter and radio transceiver). The crystal oscillator can be started and stopped as requested by software. While it is stopped, nRF9E5 uses the internal low power 4KHz RC oscillator which runs continuously (as long as 1.8V of power is supplied) and ensures that vital functions such as the wake-up timer are functioning even in deep power saving modes.

The microcontroller's architecture is 8-bit: each machine language opcode (operation code) is a single 8-bit value, which allows for 256 different instruction codes. Most of 8051's registers are 8-bit values, e.g. the Accumulator, each of the Register Banks. There are several special cases where a given register is referred to as 16-bit (such as the three Timers), but in fact these registers are addressed as two separate 8-bit registers often referred to as High and Low indicating which part of the 16-bit value they hold. The only truly 16-bit values that the 8051 handles are the Program Counter (PC) indicating the address of the next instruction to be executed and Data Pointer (DPTR) used for memory addressing. The CPU is only capable of performing basic mathematical operations on two 8-bit numbers at each cycle. There is no additional hardware support for calculation of numbers larger than 8-bit or any decryption/encryption coprocessors. In consequence manipulation of larger numbers requires numerous 8-bit calculations, for example a multiplication of two 16-bit numbers requires 9 CPU instructions.

The 8-bit word size, relatively low CPU clock frequency and the lack of mathematical hardware coprocessors in the nRF9E5 narrow the area of possible security protocols and algorithms which can be successfully implemented to those that do not require exhaustive calculations (required by most of the Asymmetric Cryptography techniques) and those that are optimized for 8-bit values. In consequence, only lightweight authentication protocols and lightweight encryption algorithms are reviewed and analysed in this dissertation.

3.5 Technical description of the memory structure and its limitations for effective security implementations

The nRF9E5 microcontroller has 256 bytes of Internal Data Memory used as a RAM with fast access, 128 Special Function Registers (one byte each) used to set different operating modes of the CPU and the Radio Frequency Transceiver. Additionally it contains 4 kilobytes of external on-chip RAM. The memory uses Harvard Architecture and is organized into six different memory spaces (see Figure 3.3). It provides 128-bytes of directly addressable DATA RAM (8052 compatible) but may also be used to hold IDATA-addressed variables. The next 128 bytes are the IDATA memory area which is accessible through indirect addressing and effectively interleaves with the Special Function Register (SFR) which in turn is directly accessed. The entire 4K of memory (addresses above 0FFh) is accessible as an external XDATA memory but this area is shared with the CODE memory, so the use of XDATA variables effectively limits the available code space. The first of 256 bytes of XDATA can be addressed in paged mode and in this configuration it is referred to as PDATA. Memory addressing diagram can be seen in Figure 3.3.

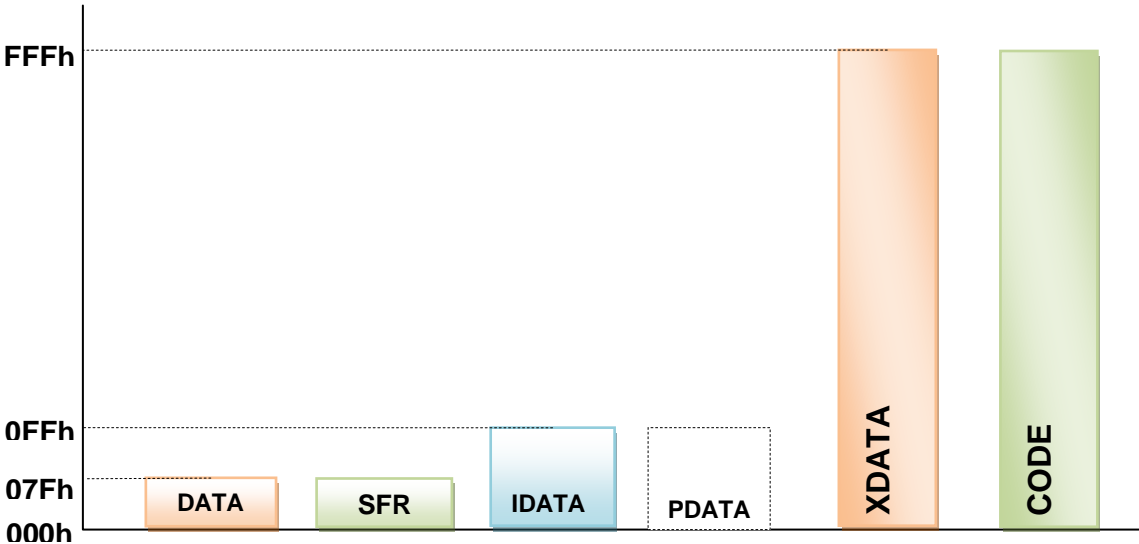


Figure 3.3 8051 Memory Addressing

The memory structure can be represented in the following manner:

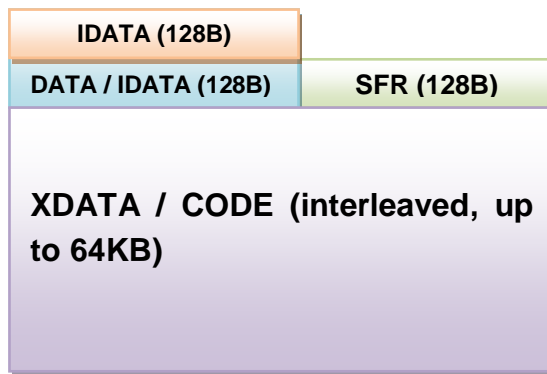


Figure 3.4 Physical organization of memory on 8051

Additionally, there is a small 512 byte ROM area located on-chip and containing bootstrap program executed automatically after power on or reset. The bootstrap loads the user program into on-chip 4K RAM from the off-chip external EEPROM memory required for operation.

The manufacturer of the chip did not include any options to extend the RAM size above the 4KB - it is not possible to connect any additional external memory directly to the CPU pins. Additionally, the bootstrap program in ROM cannot be updated. The only memory size expanding option is to use an external EEPROM memory (generic 25320 with SPI) attached through one of the GPIO pins and interfacing through a common SPI bus. Accessing external memory through the SPI bus has major consequence on the performance of the CPU as each of the SPI read/write (performed byte-by-byte) operations takes several processor cycles. The CPU is not able to perform additional tasks while in this process. One of the major limitations is the fact that external EEPROM cannot be used to expand the possible program size. In consequence, the program code size is always limited to 4KB – the bootstrap program will ignore anything above 0FFFh address in EEPROM when loading the program. Section 4 will attempt to provide a solution to overcome this limitation with the support of 8051 dedicated software Assembly Language Linker.

Possible security implementations have to be filtered through the following constraints imposed by nRF9E5:

- **Limited code/RAM space** – The CODE and XDATA space are shared in this CPU’s architecture, so variables and constants allocated here limit the overall code space. The existing Infrastructure WSN programs already occupy a vast amount of the code/RAM space (HINT Project 2010), so the algorithms/protocols have to be implementable with a minimum machine code size and there must be a limited need for variable memory allocation. In case the solutions used to overcome the memory limitations (see section 3.7) fail the space for the machine code may be limited to approximately 200 bytes

only (assuming that for an existing sensor program already occupies 95% of the available code space).

- **Extremely slow access to the external non-volatile memory** – the reference platform utilizes 32Kb 25320 generic EEPROM. The amount of the data which needs to be accessed from the external EEPROM memory and the frequency of the access has to be limited. This imposes restrictions on possible encryption key sizes and the usage of non-volatile protocol-specific data.
- **Hacking the EEPROM** - The EEPROM memory can be read by freely available EEPROM programmers, thus in the case of a physical access attack the amount of information disclosed cannot compromise the security of the entire system. This forces solutions without global pre-shared encryption keys. An example of a physical access attack compromising the security of the entire WSN using the TinySec Protocol (Karlof et al. 2004) was described in (Hartung et al. 2005).

3.6 Technical description of the radio transceiver and its limitations for effective security implementations

The nRF9E5 single chip microcontroller integrates a nRF905 (Nordic Semiconductors 2009a) compatible Radio Frequency (RF) transceiver operating on 433/868/915MHz bands (sub-1GHz). The transceiver consists of a fully integrated frequency synthesizer, a power amplifier, a modulator and receiver chain with demodulator.

The modulation type used in nRF905 is Gaussian Frequency Shift Keying (GFSK) with a data rate of 100kbps. The data bits are encoded and decoded using Manchester Encoding/Decoding and the effective symbol rate is limited to 50kbps (one symbol per two clock signals); however, no scrambling on the microcontroller is needed.

The transceiver uses SPI bus for reprogramming and data input/output. It is equipped with a circuit able to calculate the Cyclic Redundancy Check (CRC) checksum of the incoming or outgoing data packets. Transmitting (TX) and Receiving (RX) addresses can be 1 to 4-byte long and the data payload length may vary from 1 to 32 bytes.



Figure 3.5 NRF9E5 packet structure

Each data packet contains the following (see Figure 3.5):

- Preamble - predefined 10-bit sequence used to adjust the receiver for optimal performance.
- TX Address - programmable recipient's address with a length of 1 to 4 bytes.
- Payload - user data, length of the field configurable within 1 to 32 bytes range.

- CRC - 8 or 16-bit CRC checksum.

During the TX mode the packet is assembled automatically by the transceiver once the Payload and TX address is supplied – the CRC is calculated and added with a Preamble. After a transmission the RF transceiver sets the Data Ready (DR) pin high, so the microprocessor can be notified of a finished transmission.

In RX mode the radio is used to listen for incoming transmissions and if one occurs the Carrier Detect (CD) pin is set high. After this action the nRF905 analyses the Address field and discards the packet if it is destined for a different address or accepts it if the address matches, sets the Address Match (AM) pin high, reads in the payload to the buffer and verifies the CRC checksum.

The way the RF transceiver handles incoming packet addressing (automatic packet discarding when the address does not match) imposes constraints on the possibilities of protection against traceability (ID disclosure related) attacks (Juels 2006). In consequence in a situation where the communication is initiated by the Master device the packet will need to hold a broadcast address and all Slaves should be able to temporarily reconfigure themselves to accept such packets. A frequent usage of broadcast addressing may negatively impact the performance of the entire network (Ni et al. 1999). Another solution would require Slaves to ignore address mismatch and examine each packet which again reduces the performance of the network.

The maximum Payload size of 32 bytes seems large but the bandwidth of only 50kbps has to be taken into consideration too. In the presence of multiple devices operating on the same frequency the transmission time has to be limited to avoid network congestion. It has to be noted also that the entire NRF9E5 consumes the highest possible amount of power during radio transceiver operations (up to 30mA at 10dBm output power comparing to 2.2mA when only the 8051 CPU is active), thus large data transfer, although possible, can severely degrade the sensor's lifetime.

In consequence of the above limitations, the security system has to impose low radio bandwidth requirements.

3.7 Overcoming limitations: Code Banking on the nRF9E5

The major limitations of the reference platform nRF9E5 chip are the code space size and the lack of any coprocessors enhancing mathematical calculations. While the latter can be overcome by using less CPU intensive security protocols and algorithms, the program size and RAM limitations are hard to overcome without changing the entire microcontroller platform. A software solution to this issue using the concept of Code Banking with the native support of the 8051 Assembler Linker (similar solutions are available from KEIL (ARM Ltd. 2009a) and Raisonance (Raisonance SAS 2010) Integrated Development Environments) is proposed below.

The origin of the Code Banking concept (ARM Ltd. 2009b) comes from the 16-bit memory addressing limitation of the 8051 CPU. Due to the addressing bit width the

maximum memory which can be allocated is limited to 64Kb. The Code Banking mechanism permits and increase in the Code memory size up to 1MB (KEIL linker) or 4MB (Raisonance linker) by splitting the program into a Common Area section and a number of memory banks (see Figure 3.6). The Common Area (of a user-defined size *s*) and one of the Code Banks is loaded at a given time, so the microcontroller can effectively “see” and address 64KB of the Code memory. If a function makes a call to another function the linker generates a code performing that switch, called a Bridge. All bridges are located in the Common Area which remains the same regardless of which Code Bank is currently used. The full description of the assembly language routines performing bank switches and limitations such as interrupt vector handling are outside the scope of the document and can be found in Raisonance and KEIL linkers’ documentation (ARM Ltd. 2009b).

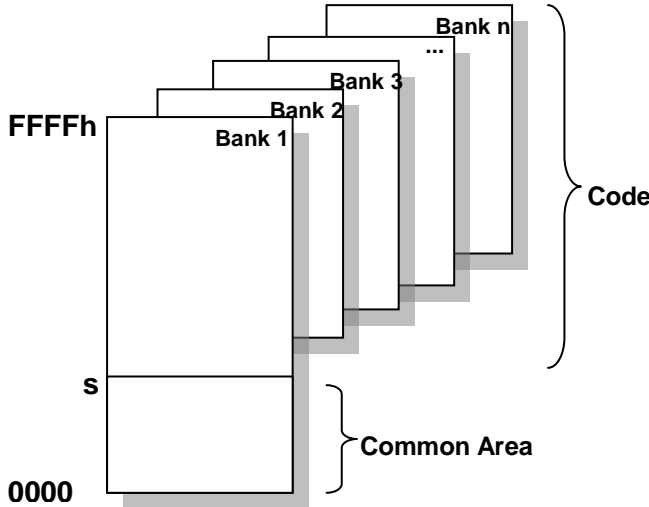


Figure 3.6 Code Banking Layout

A typical hardware design scenario permits connecting the memory directly to the CPUs I/O ports. In this case a bank switch process would only require changing the input/output port number to access different blocks of memory, where additional code banks are located (see Figure 3.7). The Common Area has to be duplicated across all memory blocks so it would still be accessible in the same form after changing the I/O ports.

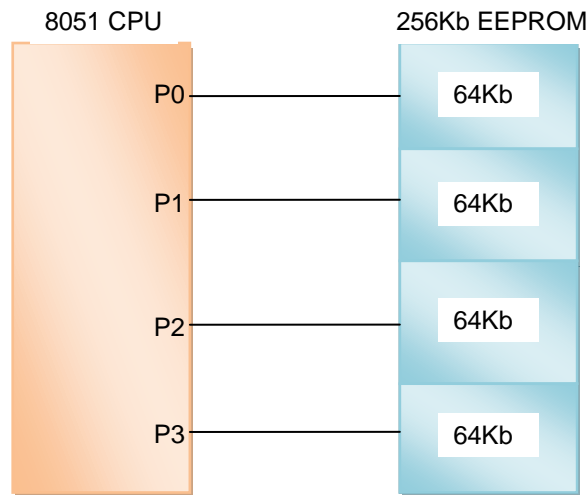


Figure 3.7 8051 with 156Kb EEPROM attached to ports P0-P3

In case of the reference platform with the nRF9E5 microcontroller, where it is not possible to connect any additional memory directly to the CPU, the Code Banking mechanism can be utilized to overcome the Code space limitation but in a manner different to the original design. Instead of using the directly attached memory chip an external EEPROM connected to the SPI bus can be utilized to hold additional code banks. However, the I/O pin switching routine has to be replaced with a function that overwrites the code bank space in the on-chip RAM with the content of this bank located on the external EEPROM. Every bank switch will be a very slow process since the entire code bank binary file (2-3Kb) has to be read through the SPI bus from the external EEPROM (see Figure 3.8). Initial experiments performed by the HINT Project team (HINT Project 2010) proved that it takes 65 milliseconds to load a Code Bank of 2Kb in size.

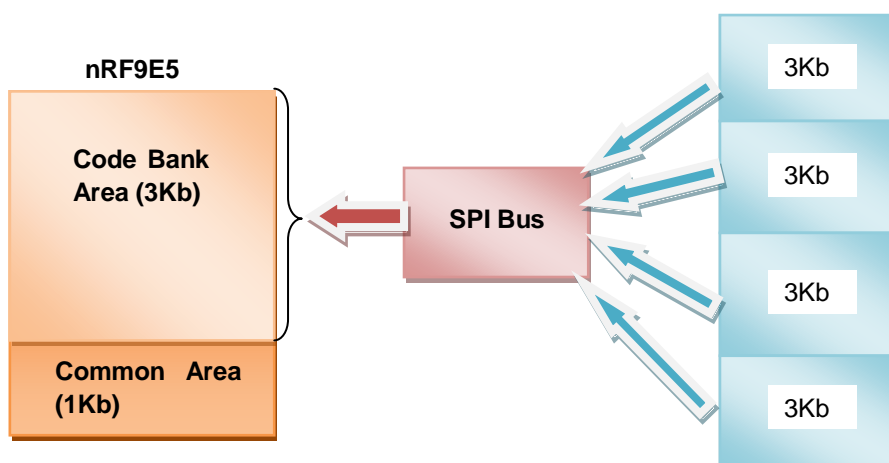


Figure 3.8 nRF9E5 code banking with an external SPI-accessed EEPROM

Despite the negative effect on the microcontroller's performance this mechanism permits the effective expansion of the available code space above 4Kb without any

hardware modifications in the existing reference platform. This would allow providing a relatively large code space for the implementation of the security mechanism. The failure of this concept would result in significant code space limitations for the security algorithms and force the usage of slow –access EEPROM-located variables.

4. Implementation

4.1 Hardware-related requirements for the implementation

The main development platform used is the Nordic Semiconductors nRF9E5 microcontroller that is used for both master and slave devices. The nRF9E5 has limited resources (see Section 3.4 for details) and this has implications for the implementation of authentication and encryption on these devices. This limitation is somewhat eased by (a) offloading computationally heavy tasks from the master to a back-end server, allowing the master to more effectively handle service requests from slave devices and (b) by improving code memory space using code banking. Neither of these enhancements has been used in this project. Since a vastly scaled down communication and radio protocol is used, the inherent memory of the nRF9E5 is sufficient to effectively run the security mechanisms. Variables that would otherwise be serviced from a back-end server have been hard-coded into master and slave, negating the use of the back-end server in the developed prototype. However, in a field implementation of secure sensor networking, where many slaves communicate with a master, it would be necessary to use a back-end server and overcome the code space limitations through code banking. Considering the limitations of the devices, a C language implementation was chosen instead of 8051 native Assembly code to allow faster porting to other platforms.

The main limitation of the nRF9E5 microcontroller in terms of the implementation was the maximum code size of only 4 kilobytes. The prototype was implemented to fit under this barrier. However, some protocol simplifications were needed to achieve small code space. These simplifications are further described in section 4.4. The amount of RAM (256 bytes for both Data and Idata) was sufficient but almost entirely used by both master and slave prototypes.

The radio transceiver embedded on nRF9E5 requires pre-configuration and manual handling of the OSI Model Data Link and upper layers. This generates another code space requirement, thus a simplified radio protocol is used in the prototype. The hardware design of the radio transceiver offers two useful tools that simplify the radio protocol implementation: Address Match and Carrier Detect bits. These tools were used to implement a simple Listen-Before-Talk collision avoidance scheme.

4.2 Integrated Development Environment (IDE) and Hardware utilised.

There are two well known Integrated Development Environments offering packaged Assembler and ANSI-C compilers for the 8051-compatible microcontrollers: KEIL (ARM Ltd. 2009a) and Raisonance RC51 IDEs (Raisonance SAS 2010). Raisonance RKit Eval51 was utilised as it offers an 8051 compiler fully functional with the exception of a code size limited to 4 kilobytes. The code size limitation perfectly matches the hardware limitation of the nRF9E5 microcontroller.

The hardware used in the implementation stage were two Nordic Semiconductor Evaluation Boards nRF9E5-EVBOARD with EEPROM emulator/programmer USB dongles nRF24E1. The programming dongles were controlled by the nRFPROG software supplied by Nordic Semiconductors.

4.3 Design - algorithms for both authentication and encryption

The prototype is designed to fit within 4 kilobytes of total code space available for programs on the nRF9E5 reference platform. The usage of code banking or other techniques overcoming the 4KB limitation is not considered in the prototype implementation. Instead some minor simplifications in the protocol (explained below) are used. The scope of the prototype is explained in Figure 4.1. The back-end database and PC Host software are outside the scope of the implementation – it will focus only on the 8-bit microcontroller code written in the C language with nRF9E5-specific radio transceiver handling functions.

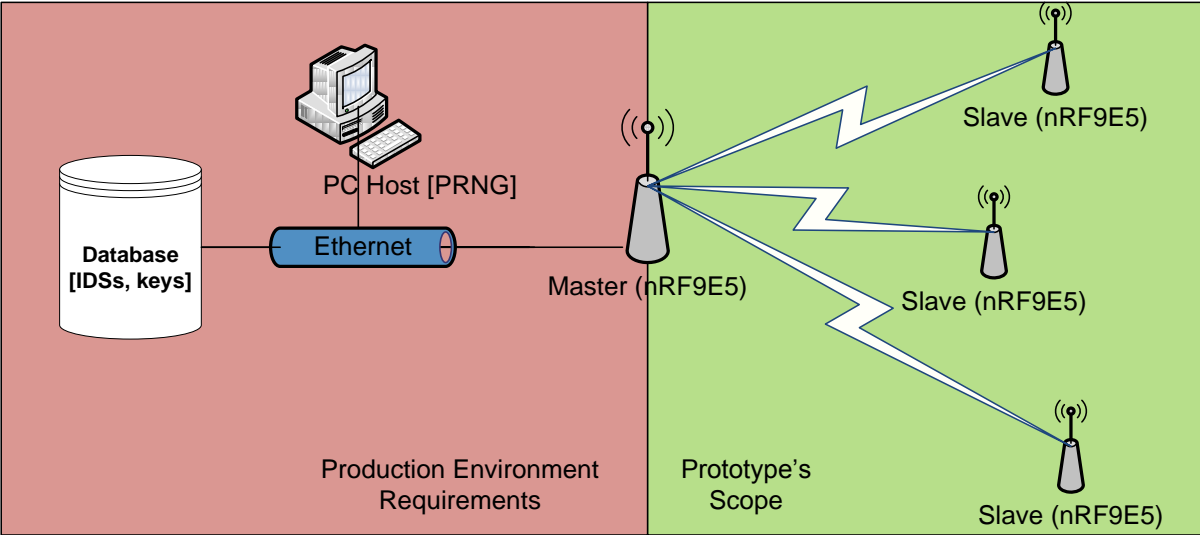


Figure 4.1 The Scope of the Implementation part

Radio Protocol

The prototype uses a simplified radio protocol allowing communication between the Master and the Slave. Both devices utilise Address Match (AM) and Carrier Detect (CD) bits. The Carrier Detect bit will be used to implement a simple Listen-Before-Talk collision avoidance mechanism. Both devices test the CD bit before switching the radio into transmitting (TX) mode. In this simplified model a transmitting device will loop forever waiting for the CD bit to be clear and attempt the transmission straight away after this bit is cleared. Due to code space constraints random TX back-off period (Pure-Aloha Protocol) or wait-until-CD timeout is not implemented in the prototype.

Authentication

The Gossamer lightweight authentication protocol (see Section 2.2.3 for full description) was chosen to fulfil the requirement for mutual authentication between the Master and the Slave devices (see Section 0) in Infrastructure WSN (see Section 3.1). This protocol was chosen mainly thanks to its proven security, low memory and computation requirements, and the expected simplicity of the implementation of all necessary mathematical operations on 8051-compatible CPU.

The original design of the Protocol is simplified for the prototype implementation purposes in the following areas:

- The keys and IDS are not stored persistently on the Slave device due to code space overhead imposed by the EEPROM read/write routines. Upon each power loss these values will be reset to the initial ones.
- Master side: random numbers n_1 and n_2 will be replaced by hard-coded values for experimentation purposes. The IDS of a sample Slave will also be hard-coded, so the back-end database will not be needed in the simplified model.
- Slave side: in the original Gossamer Protocol the Slave device sends the value D but there is no acknowledgement that D has been received and verified by the Master. The Slave then updates its keys and IDS and saves the previous IDS and key values. In a subsequent round, if the slave cannot verify value C , in which case authentication of the master will not have been a success, the slave can roll back to the previous keys and IDS values. This de-synchronization attack prevention mechanism has not been implemented in the simplified protocol.

Figure 4.2 shows the full round of the Gossamer authentication protocol adapted to the needs of the Infrastructure WSN. The main difference was the removal of the 'Hello' message as in IWSN the Slave device (Tag equivalent in standard Gossamer specification) initiates the communication.

Encryption

The Scalable Encryption Algorithm (SEA - see Section 2.3.3 for full description) was chosen as the encryption mechanism. SEA(96, 8) mode was used, meaning that the block and the key size of 96-bits and 8-bit word matching the word size on the nRF9E5. The choice of the algorithm can be justified by the lack of proven weaknesses in the algorithm and the fact that the algorithm can be implemented with a very limited code space by sacrificing the throughput of the encryption (number of words that can be encrypted over a given period of time). The reduced throughput of the algorithm is not a significant issue in the context of IWSN, where the amount of data transferred is very small in most cases.

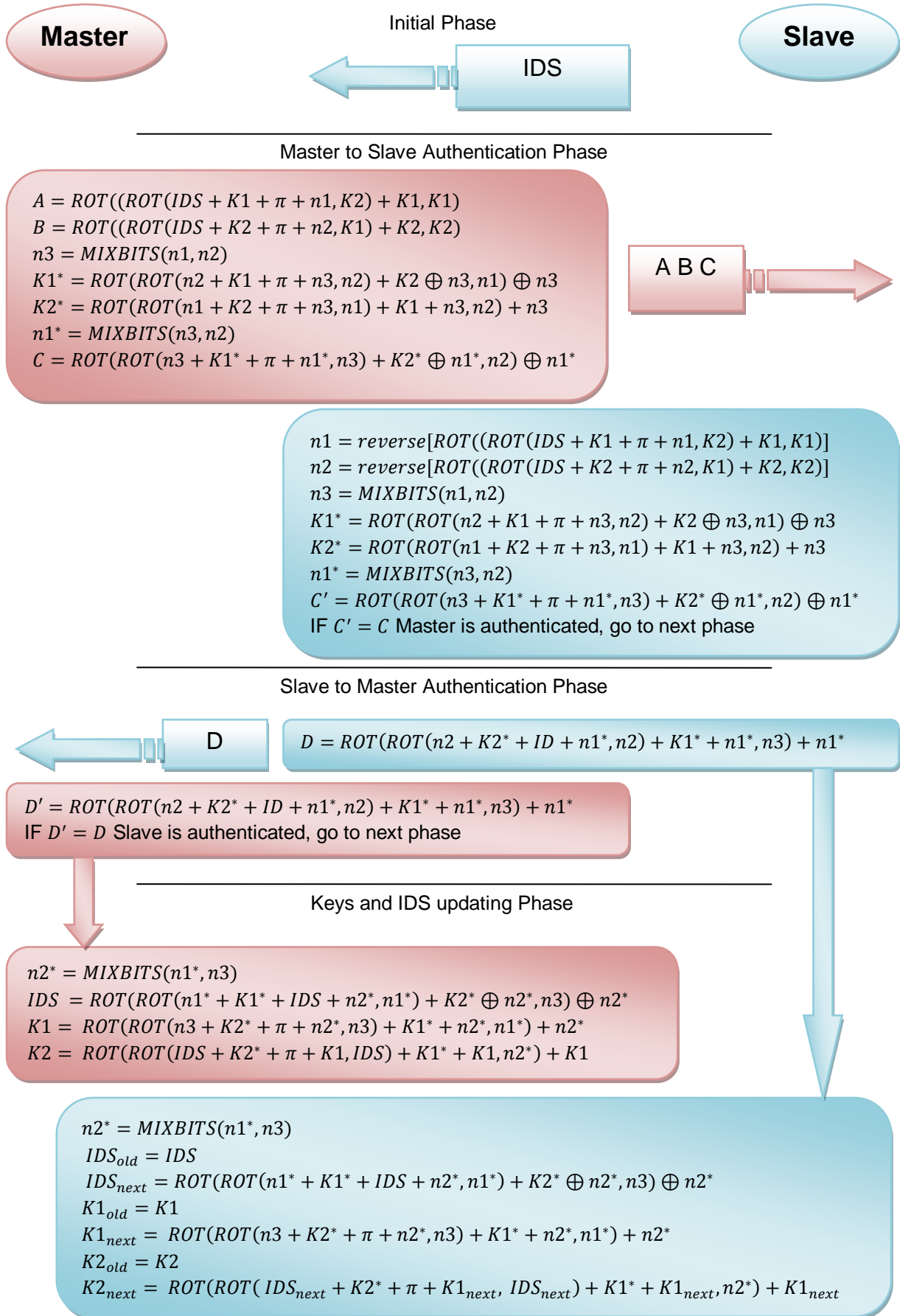


Figure 4.2 Gossamer Protocol Adapted to the Infrastructure WSN

Another advantage of this algorithm is its scalability which permits increasing key and word sizes to 192 bits without major modifications of the code. This can be applied in cases where the 96-bit security is not regarded as strong enough.

Since the Gossamer authentication protocol exchanges two new 96-bits keys at each round, one of these keys can be used as an encryption key for the SEA(96,8) algorithm during one communication session between the Master and the Slave.

4.4 Coding - Main elements of code explained

Both the Master and the Slave programs were written in two separate modules: Master.c and Slave.c. Each of the modules contains the following main elements:

- Initialization (UART timers, radio),
- Utilities Block (UART handling, SPI handling),
- Radio Handling Block (TX and RX),
- Gossamer functions,
- SEA functions.

The organization of the main module for both Master.c and Slave.c is explained in Figure 4.3. The full code can be found in Appendix B. This sub-section describes the Gossamer and SEA function.

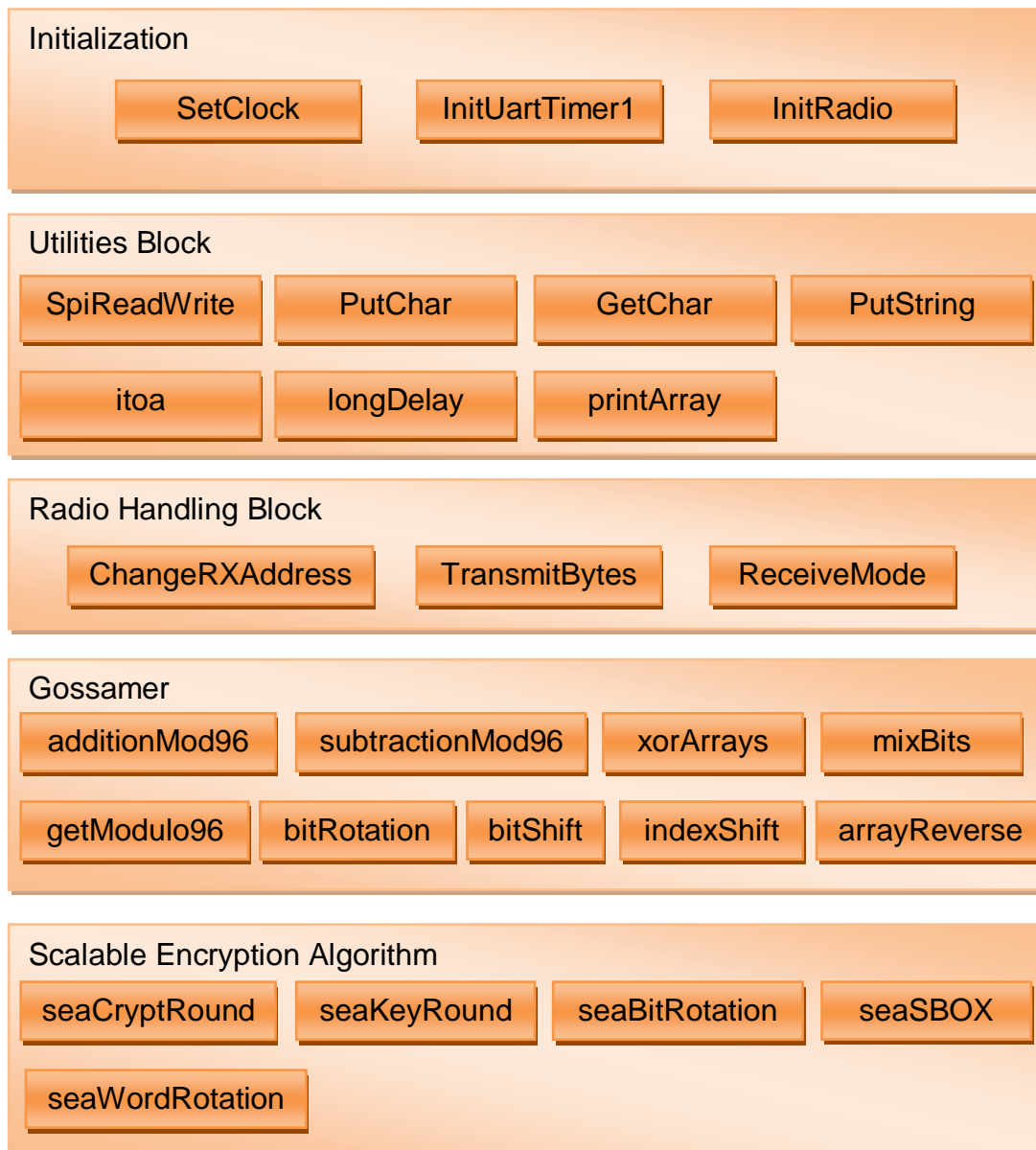


Figure 4.3 Main Program Components

4.4.1 Gossamer Implementation

The Gossamer Protocol implementation uses the main gossamerMaster and gossamerSlave loops following the design explained in Figure 4.2. All 96-bit values are implemented as an array of 12 unsigned characters (one byte each) in Big-endian (Most Significant Bit first) notation. The IDS, ID, K1, K2 and Pi values are initialized on the startup of the main loop, thus on every power-loss they are reset to the hard-coded values.

All Gossamer-Related mathematical operations are implemented in separate functions explained below.

Addition Modulo96

Performs addition Modulo96 on two arrays passed as parameters and saves the result into the second argument's memory location. The function simply adds each element in the array one-by-one starting with the last element (12) and carries a bit over to the lower element if the result is larger than 255. If the lower elements are 255 already then the bit is carried over to lower elements until the head of the array if needed.

```
void additionMod96 (unsigned char idata *array1, unsigned char idata
*result)
{
    unsigned char i;
    unsigned char j;

    for (i=11; i>0; i--)
    {
        result[i] += array1[i]; //Add two bytes (no carry)
        if (result[i] < array1[i]) //Check if carry needed and append to
            //upper byte
        {
            result[i-1]++;
            //check if previous byte was not 255 overloaded to 0 and step
            //back to lower elements to do the same
            j=i;
            //If a carry bit overloads upper byte increment upper to
            //overloaded
            //Continue until the array head is met if needed
            while(result[j-1] == 0 && j > 1)
            {
                result[j-2]++;
                --j;
            }
        }
    }
    result[0] += array1[0]; //Got to the MSB - just add and ignore carry
}
```

Figure 4.4 Code: Addition Modulo96

Subtraction Modulo96

Performs subtraction Modulo96 on two arrays passed as parameters and saves the result into the second argument's memory location. Similarly to the Addition function it simply subtracts each element one-by-one starting with the last element. If the minuend is smaller than the subtrahend, then a bit is borrowed from the lower element. If the lower elements are zeros then the borrow bit is taken from lower elements until the array head is met. This function is only required on the Slave side as it is only needed when extracting random numbers n1 and n2 from message A and B respectively.

```
void subtractionMod96 (unsigned char idata *array1, unsigned char idata
*result)
{
    unsigned char i;
    unsigned char j;
```

```

for (i=11; i>0; i--)
{
    if (result[i] < array1[i]) //Verify if the minuend is not smaller
                                //than the subtrahend
    {
        result[i-1] -= 0x01; //borrow LSB from the lower element
        j=i;
        //If a borrow bit overloads upper byte decrement upper byte to
        //the overloaded one
        while(result[j-1] == 0xFF && j > 1)
        {
            result[j-2] -= 0x01;
            j--;
        }
    }
    result[i] -= array1[i]; //Subtract (no carry)
}
result[0] -= array1[0]; //Got to the MSB - just add and ignore carry
}

```

Figure 4.5 Code: Subtraction Modulo96

XOR two 96-bit numbers

This function loops through all elements in the array and performs a bitwise exclusive OR operation on them one-by-one.

```

void xorArrays (unsigned char idata *array1, unsigned char idata *result)
{
    unsigned char i;

    for (i=0; i<12; i++)
    {
        result[i] ^= array1[i];
    }
}

```

Figure 4.6 Code: XOR two 96-bit numbers

Bit Rotation (ROT) on two numbers

The bitRotation function performs circular bit rotation of a 96-bit number by a Modulo96 of a second number passed as a second parameter. This function uses four sub-functions to perform the bit rotation:

- getModulo96 - returns Modulo96 of a 96-bit number passed in the array of 12 one-byte elements. The function uses a command and conquer approach. Starting from the lowest element a Modulo96 of each element (split into two 4-bit numbers and multiplied by 256) is calculated one-by-one and added to the overall result. At each iteration the overall result is reduced Modulo96.

```

unsigned char getModulo96 (unsigned char idata *array)
{
    unsigned char i;
    unsigned char modulus = 0;
}

```

```

    for (i=0; i<11; i++)
    {
        //Divide and conquer approach: sum of two 4-bit numbers multiplied
//by 256
        modulus += (( (array[i] & 0x0f) + (array[i] >> 4) ) * 256) % 96;
        modulus %= 96; //Reduce each result Mod96 - can be done less
//frequently
    }
    return (array[11] + modulus) % 96; //Add the result to the LSB and
//calculate Mod96 again
}

```

Figure 4.7 Code: Get Modulo96

- bitShift - performs circular bit-shift (up to 7 places) of each element in the array in both directions. Depending on the direction the remainder of the shift is appended to the lower or upper element.

```

void bitShift (unsigned char idata *array, unsigned int direction, unsigned
int bitsToShift)
{
    unsigned char i;
    unsigned char element0;
    unsigned char temp;

    //Direction: 0 for left shift, 1 for right shift
    if (!direction) //Shift bits to the left with carry to the lower
element
    {
        element0 = array[0];
        array[0] = array[0] << bitsToShift;

        for (i=0; i<11; i++)
        {
            temp = array[i+1];
            array[i+1] = array[i+1] << bitsToShift;
            array[i] |= temp >> (8 - bitsToShift);
        }
        array[11] |= element0 >> (8-bitsToShift);
    }
    else //Shift bits to the right with carry to the lower element
    {
        element0 = array[11];
        array[11] = array[11] >> bitsToShift;

        for (i=11; i>0; i--)
        {
            temp = array[i-1];
            array[i-1] = array[i-1] >> bitsToShift;
            array[i] |= temp << (8 - bitsToShift);
        }
        array[0] |= element0 << (8-bitsToShift);
    }
}

```

Figure 4.8 Code: Bit Shift

- indexShift - rotates the elements of the array by up to 11 positions left or right. It takes advantage of the arrayReverse function and a formula assuming that the array is split into two sub-arrays A and B (A.B), where the size of array A is the number of places the elements are to be rotated. The formula is as follows: B.A = reverse(reverse(A).reverse(B)).

```
void indexShift (unsigned char idata *result, unsigned int direction,
unsigned int indexShift)
{
    //Direction: 0 for left shift, 1 for right shift
    if (!direction)
    {
        arrayReverse(result, 0, indexShift-1);
        arrayReverse(result, indexShift, 11);
        arrayReverse(result, 0, 11);
    }
    else
    {
        arrayReverse(result, 12-indexShift, 11);
        arrayReverse(result, 0, 11-indexShift);
        arrayReverse(result, 0, 11);
    }
}
```

Figure 4.9 Code: Index Shift

- arrayReverse - reverses the elements in the array (array[beginning] becomes array[end] and so on).

```
void arrayReverse(unsigned char idata *result, unsigned char left, unsigned
char right)
{
    unsigned char temp;
    unsigned char i;
    unsigned char j;

    //Start with edges and continue until middle elements are processed
    for (i=left, j=right; i<j; i++, j--)
    {
        temp = result[i];
        result[i] = result[j];
        result[j] = temp;
    }
}
```

Figure 4.10 Code: Array Reverse

The bitRotation function calculates the Modulo96 of the first argument (array to be rotated by) and then analyses the result to verify if bitShift and indexShift functions need to be called and calls them accordingly.

```
void bitRotation (unsigned char idata *array1, unsigned char idata *result,
unsigned int direction)
{
    unsigned char modulo = getModulo96(array1); //First get modulo
    unsigned char indicesToShift;
    unsigned char bitsToShift;
```

```

//Second divide modulo by 8 and rotate the array (if modulo is bigger
//than 8)
//Direction: 0 for left shift, 1 for right shift
if (modulo > 8)
{
    indicesToShift = modulo/8;
    indexShift(result, direction, indicesToShift);
}

//Then bitshift with carry each element by the remaining shift (shift
//amount will be <8)
bitsToShift = modulo%8;
if (bitsToShift != 0)
    bitShift(result, direction, bitsToShift);
}

```

Figure 4.11 Code: Bit Rotation

MixBits function

The MixBits function implements the Gossamer author's recommendation shown in Figure 4.12.

```

Z = MixBits (X, Y)
Z = X
FOR counter = 0 to 32
Z = (Z>>1) + Z + Z + Y
ENDFOR

```

Figure 4.12 MixBits Function pseudocode

The function uses two arrays passed as parameters and a temporary array returned with the result. Functions described above (additionMod96 and bitShift) are utilized.

```

unsigned char* mixBits (unsigned char idata *array1, unsigned char idata
*array2)
{
    // Z = mixBits (X,Y)
    unsigned char idata result[12];
    unsigned char i;

    // Z = X
    for (i=0; i<12; i++)
    {
        result[i] = array1[i];
    }
    // 32times: Z = (Z>>1) + Z + Z + Y
    for (i=0; i<32; i++)
    {
        bitShift (array1, 1, 1);
        additionMod96 (array1, result);
        bitShift (array1, 0, 1);
        additionMod96 (array1, result);
        additionMod96 (array1, result);
        additionMod96 (array2, result);
    }
    return result;
}

```



```
}
```

Figure 4.13 Code: MixBits

The main Gossamer loop following the procedures listed in Figure 4.2 can be found in Appendix B.

4.4.2 Scalable Encryption Algorithm (SEA) Implementation

The SEA(96, 8) implementation uses a word size of 8-bits (unsigned char) with a block and key size of 96-bits. Both are passed as an argument in a form of a 12-element array of unsigned characters. The main components of the SEA implementation are the following functions: cryptographic round, the key round, the S-Box, the bit-rotation, the word-rotation and the main SEA wrap-up function. In this prototype the key used in encryption will be either k1 or k2 updated by the Gossamer function at each authentication round.

SEA Substitution Box

Per SEA author's suggestions the S-Box can be applied bitwise to any 3 elements of a block-half currently being processed (for blocks of 96-bits). Since there are 6 one-byte elements in each half of the block the S-Box can be applied on two different set of words. (Standaert et al. 2006) suggested a function ('i' equals 0 or 1) shown in Figure 4.14.

```
void seaSBOX (unsigned char data *block, unsigned char i)
{
    block[3*i] = (block[3*i+2] && block[3*i+1]) ^ block[3*i];
    block[3*i+1] = (block[3*i+2] && block[3*i]) ^ block[3*i+1];
    block[3*i+2] = (block[3*i] || block[3*i+1]) ^ block[3*i+2];
}
```

Figure 4.14 Code: SEA S-Box

Standaert agrees that it is safe to simplify this function so the S-Box is only to the first three elements in order to reduce the code space required by this function.

```
void seaSBOX (unsigned char idata *block, unsigned char i)
{
    block[0] = (block[2] && block[1]) ^ block[0];
    block[1] = (block[2] && block[0]) ^ block[1];
    block[2] = (block[0] || block[1]) ^ block[2];
}
```

Figure 4.15 Code: SEA S-box modified

SEA Bit Rotation

The Bit Rotation function in SEA(96,8) implementation performs circular bit-rotation one place to the right on words numbered 0 and 3, and one place to the left on words numbered 2 and 5. The seaBitRotation function implemented uses Reasonance RC51 compiler's intrinsic functions '_crol_' and '_crol_' to save the code space.

```

void seaBitRotation (unsigned char idata *block)
{
    block[0] = _cror_(block[0], 1);
    block[2] = _crol_(block[2], 1);
    block[3] = _cror_(block[3], 1);
    block[5] = _crol_(block[5], 1);
}

```

Figure 4.16 Code: SEA Bit-Rotation

SEA Word rotation

The seaWordRotation function performs circular right- or left-rotation of the block-half array elements by one place. The Gossamer indexShift function can be re-used to save the code space but this function was also implemented to make the SEA module independent and re-usable without the Gossamer functions overhead.

```

void seaWordRotation (unsigned char idata *block, unsigned char direction)
{
    //Direction 0 for left and 1 for right rotation
    unsigned char i;
    unsigned char temp;

    if (direction == 0)
    {
        temp = block[0];
        for (i=0; i<5; i++)
            block[i] = block[i+1];
        block[5] = temp;
    }
    else
    {
        temp = block[5];
        for (i=5; i>0; i--)
            block[i] = block[i-1];
        block[0] = temp;
    }
}

```

Figure 4.17 Code: SEA Word-Rotation

SEA Encrypt/Decrypt round

The seaCryptRound function performs one round encryption or decryption round using left and right half of the block and one half of the key - left or right depending on the round. This function implements the following SEA equations: F_E encryption function and F_D decryption function (below). See Figure 2.6 for a graphical representation of the SEA encryption/decryption round.

$$F_e(L_i, R_i, \text{KeyHalf}) = \text{RightWordRot}(L_i) \text{ XOR } \text{bitRotation}(\text{sbox}(R_i + \text{KeyHalf}))$$

$$F_d(L_i, R_i, \text{KeyHalf}) = \text{LeftWordRot}(L_i \text{ XOR } \text{bitRotation}(\text{sbox}(R_i + \text{KeyHalf})))$$

The function takes advantage of previously described word rotation, bit rotation and substitution box functions.

```

void seaCryptRound (unsigned char direction, unsigned char idata
*blockLeft, unsigned char idata *blockRight, unsigned char idata *keyHalf)
{
    unsigned char i;
    unsigned char temp[6];

    //Every operation will be performed on blockLeft as this memory
    //location will become the right block for the next round.
    //Save the right block
    for (i=0; i<6; i++)
        temp[i] = blockRight[i];

    //ENCRYPTION
    //Fe(Li, Ri, K/2i) = RightWordRot(Li) XOR bitRot(sbox(Ri+K/2i))
    //DECRYPTION
    //Fd(Li, Ri, K/2i) = LeftWordRot(Li XOR bitRot(sbox(Ri+K/2i)))

    //Step by step:
    //Ri+K/2i
    for (i=0; i<6; i++)
        blockRight[i] += keyHalf[i];
    //sbox(Ri+K/2i)
    seaSBOX(blockRight, i%2);
    //bitRot(sbox(Ri+K/2i))
    seaBitRotation(blockRight);
    //RightWordRot(Li) - encryption only
    //Direction 0 for encryption and 1 for decryption
    if (direction == 0)
        seaWordRotation(blockLeft, 1);
    //RightWordRot(Li) XOR bitRot(sbox(Ri+K/2i))
    for (i=0; i<6; i++)
    {
        blockRight[i] ^= blockLeft[i];
        blockLeft[i] = temp[i]; //BlockLeft(i)+1 becomes BlockRight(i)
    }
    //LeftWordRot(Li XOR bitRot(sbox(Ri+K/2i))) - decryption only
    if (direction == 1)
        seaWordRotation(blockRight, 0);
}

```

Figure 4.18 Code: SEA Encrypt/Decrypt Round

SEA Key Round

The seaKeyRound function performs one round of the key scheduling. These rounds are interleaved with encryption/decryption rounds. Each key round performs the following key scheduling function (see Figure 2.6 for a graphical representation):

$$F_k(K_{Li-1}, K_{Ri-1}, C_i) \Leftrightarrow K_{Ri} = K_{Li-1} \text{ XOR } \text{RightWordRot}(\text{bitRot}(\text{sbox}((K_{Ri-1}) + C_i)))$$

The function takes advantage of previously described word rotation, bit rotation and substitution box functions.

```

void seaKeyRound (unsigned char idata *keyLeft, unsigned char idata
*keyRight, unsigned char Ci)
{
    //Fk(KLi-1, KRi-1, Ci) <=> KRi = KLi-1 XOR RightWordRot(bitRot(sbox((KRi-
1)+Ci)));
    unsigned char i;
}

```

```

unsigned char temp[6];

//Save the left key (left key will become right after the round)
//Every operation will be performed on keyLeft as this memory location
//will become a right key for the next round.
for (i=0; i<6; i++)
    temp[i] = keyRight[i];
//Step-by-step:
//init Ci (LSW equals i)
///Ci[5] = i;
//(KRi-1)+Ci
keyRight[5] += Ci;
//sbox((KRi-1)+Ci)
seaSBOX(keyRight, (Ci%2));
////seaSBOX(keyRight, 1);
//bitRotation(sbox((KRi-1)+Ci))
seaBitRotation(keyRight);
//RightWordRot (bitRot (sbox((KRi-1)+Ci)));
seaWordRotation(keyRight, 1);
//KRi = KLi-1 XOR RightWordRot (bitRot (sbox((KRi-1)+Ci)));
for (i=0; i<6; i++)
{
    keyRight[i] ^= keyLeft[i];
    keyLeft[i] = temp[i];    //KeyLeft(i)+1 becomes KeyRight(i)
}
}

```

Figure 4.19 Code: SEA Key Round

SEA main function

The main SEA(96, 8) function takes two 12-byte parameters: block and key. (Standaert et al. 2006) advised that the minimum safe number of encryption/decryption rounds can be calculated using the following formula:

$$\frac{3n}{4} + 1 + 2 \left(\frac{n}{2b} + \left(\frac{b}{2} \right) \right), \text{ where } n = \text{plaintext size (96 bits)}; b = \text{word size (8 bits)}$$

The odd result in case of SEA(96, 8) is 93. The main function runs interleaved encryption (or decryption) and key scheduling round 46 times. After the initial 46 rounds the key halves are swapped and another further 46 rounds are executed. After the 92nd round another one encryption/decryption round runs - the key is in its final state already. It has to be noted that this final state of the key is identical to its initial state, thus no additional memory locations are needed to store a temporary key at each round. After the last round the block halves need to be swapped and the execution of the algorithm stops.

```

void sea (unsigned char direction, unsigned char idata *block, unsigned
char idata *key)
{
    //Direction 0 for encryption and 1 for decryption
    unsigned char i;

```

```

//initialization
unsigned char* idata keyLeft = &key[0];
unsigned char* idata keyRight = &key[6];
unsigned char* idata blockLeft = &block[0];
unsigned char* idata blockRight = &block[6];
unsigned char* idata tmp; //temp pointer used for swapping key
//sides
unsigned char tmp;

//First half of all rounds (93 as per author's recommendation for a
//minimum number of rounds)
for (i=1; i<47; i++)
{
    //Key scheduling
    //[KLi, KRi] = Fk(KLi-1, KRi-1, C(i));
    seaCryptRound (direction, (unsigned char idata *)blockLeft,
(unsigned char idata *)blockRight, (unsigned char idata *)keyRight);
    seaKeyRound((unsigned char idata *)keyLeft, (unsigned char idata
*)keyRight, i);
}

//End of round half - swap pointers
tmp = keyLeft;
keyLeft = keyRight;
keyRight = tmp;

//for (i=46; i<92; i++)
for (i=46; i>0; i--)
{
    //Key scheduling part 2
    //[KLi, KRi] = Fk(KLi-1, KRi-1, C(r-i));
    seaCryptRound (direction, (unsigned char idata *)blockLeft,
(unsigned char idata *)blockRight, (unsigned char idata *)keyLeft);
    seaKeyRound((unsigned char idata *)keyLeft, (unsigned char idata
*)keyRight, i);
}
seaCryptRound (direction, (unsigned char idata *)blockLeft, (unsigned
char idata *)blockRight, (unsigned char idata *)keyLeft);

//Final: switch Block halves
//indexShift (block, 0, 6); Gossamer function may be used to save space
for(i=0; i<6; i++)
{
    tmp = block[i];
    block[i] = block[i+6];
    block[i+6] = tmp;
}
}

```

Figure 4.20 Code: SEA Main Function

The experimental implementation takes Gossamer K1 key as an encryption key for the SEA algorithm. After a successful authentication round the Master encrypts a message using K1 and sends it to the Slave. The Slave decrypts the message using K1 and outputs it to the UART.

4.5 Testing

4.5.1 Testing environment

The code implemented in the course of this research was tested using the same hardware and software as in the implementation stage. During the testing stage two nRF9E5-EVBOARD development boards with nRF24E1 EEPROM programmers were used. The EEPROM programmers were connected over the USB link and the UART input/outputs from the development boards were connected through serial cables to the RS-232 ports on the development PC running Microsoft Windows XP Operating System.

Since the RC51 compiler used does not offer nRF9E5-compatible debugger, the debugging was performed on-device using manually written debug messages sent to the UART I/O.

4.5.2 One Round Step-By-Step Test

Test Procedure

The goal of this test is to verify the proper functioning of all core functions used by the Gossamer Authentication Protocol and the SEA encryption/decryption algorithm.

Both the Master and the Slave programs are pre-configured with a Gossamer Protocol test data and set to output the data at each of the modifications so that the result can be verified with a 'paper-test' (manual calculation). The integer-to-ascii (itoa) function will be employed to output the data to the UART in a human-readable form. The SEA algorithm will not be tested step-by-step due to a large number of rounds. Instead, a result of the entire encryption and decryption loop will be displayed.

Test data

Master Side:

```
unsigned char idata Pi[12] = { 0x32, 0x43, 0xF6, 0xA8, 0x88, 0x5A, 0x30,
0x8D, 0x31, 0x31, 0x98, 0xA2 };
unsigned char idata IDS[12] = { 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01,
0x01, 0x01, 0x01, 0x01, 0x01 };
unsigned char idata ID[12] = { 0x44, 0x44, 0x44, 0x44, 0x44, 0x44, 0x44,
0x44, 0x44, 0x44, 0x44, 0x44 };
unsigned char idata k1[12] = { 0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10,
0x10, 0x10, 0x10, 0x10, 0x10 };
unsigned char idata k2[12] = { 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
0x20, 0x20, 0x20, 0x20, 0x20 };
unsigned char idata n1[12] = { 0x22, 0x22, 0xFF, 0xFF, 0x22, 0x22, 0x22,
0x22, 0x22, 0x22, 0x22, 0x22 };
unsigned char idata n2[12] = { 0x23, 0x23, 0x00, 0x00, 0x23, 0x23, 0x23,
0x23, 0x23, 0x23, 0x23, 0x23 };
```

Figure 4.21 Code: Master Side Test Data

Slave side:

```

unsigned char idata Pi[12] = { 0x32, 0x43, 0xF6, 0xA8, 0x88, 0x5A, 0x30,
0x8D, 0x31, 0x31, 0x98, 0xA2 };
unsigned char idata IDS[12] = { 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01,
0x01, 0x01, 0x01, 0x01, 0x01 };
unsigned char idata ID[12] = { 0x44, 0x44, 0x44, 0x44, 0x44, 0x44, 0x44,
0x44, 0x44, 0x44, 0x44, 0x44 };
unsigned char idata k1[12] = { 0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10,
0x10, 0x10, 0x10, 0x10, 0x10 };
unsigned char idata k2[12] = { 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
0x20, 0x20, 0x20, 0x20, 0x20 };

```

Figure 4.22 Code: Slave Side Test Data

After the Gossamer round the Master will use the modified key k1 to encrypt a message (temp array) and send to the Slave. After successful transmission the Slave will use modified k1 to decrypt the message and display it.

Test Results

The test was split into several stages to allow better readability.

Stage 1: Messages A and B creation (Master side - below).

$$A = ROT((ROT(IDS + K1 + \pi + n1, K2) + K1, K1))$$

$$B = ROT((ROT(IDS + K2 + \pi + n2, K1) + K2, K2))$$

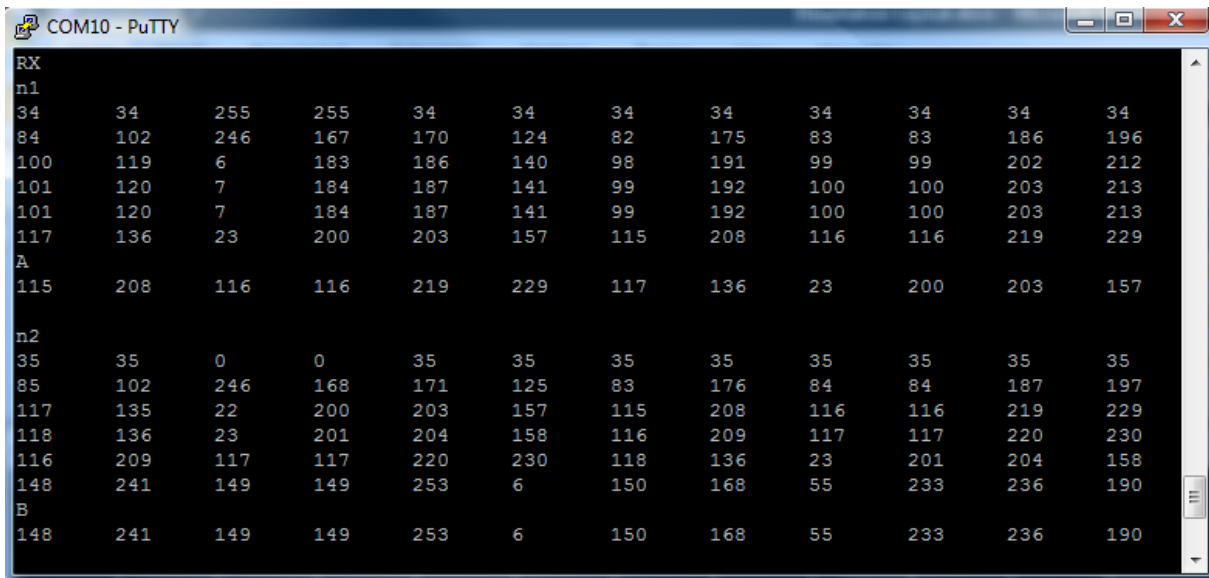


Figure 4.23 Gossamer messages A and B creation (Master).

Stage 2: N1 and N2 extraction from messages A and B (Slave side).

$$n1 = reversedA$$

$$n2 = reversedB$$

```

COM11 - PuTTY
RX
A
115 208 116 116 219 229 117 136 23 200 203 157

RX
B
148 241 149 149 253 6 150 168 55 233 236 190
117 136 23 200 203 157 115 208 116 116 219 229
101 120 7 184 187 141 99 192 100 100 203 213
101 120 7 184 187 141 99 192 100 100 203 213
100 119 6 183 186 140 98 191 99 99 202 212
84 102 246 167 170 124 82 175 83 83 186 196

n1
34 34 255 255 34 34 34 34 34 34 34 34
148 241 149 149 253 6 150 168 55 233 236 190
116 209 117 117 220 230 118 136 23 201 204 158
118 136 23 201 204 158 116 209 117 117 220 230
117 135 22 200 203 157 115 208 116 116 219 229
85 102 246 168 171 125 83 176 84 84 187 197

n2
35 35 0 0 35 35 35 35 35 35 35 35

```

Figure 4.24 Gossamer n1 and n2 random numbers extraction (Slave).

Stage 3: n3, k1next and k2next creation (Master side)

$$n3 = MIXBITS(n1, n2)$$

$$K1^* = ROT(ROT(n2 + K1 + \pi + n3, n2) + K2 \oplus n3, n1) \oplus n3$$

$$K2^* = ROT(ROT(n1 + K2 + \pi + n3, n1) + K1 + n3, n2) + n3$$

```

COM10 - PuTTY
MixBits
49 114 255 190 49 49 49 49 49 49 49 34
49 114 255 190 49 49 49 49 49 49 49 34
99 182 246 102 185 139 97 190 98 98 201 196
115 199 6 118 201 155 113 206 114 114 217 212
150 234 6 118 236 190 148 241 149 149 252 247
101 244 167 140 172 175 231 188 183 80 51 183
134 20 199 172 204 208 7 220 215 112 83 215
183 102 56 18 253 225 54 237 230 65 98 245
247 132 219 183 153 5 139 214 221 152 224 75

k1next
198 246 36 9 168 52 186 231 236 169 209 105

49 114 255 190 49 49 49 49 49 49 49 34
99 182 246 102 185 139 97 190 98 98 201 196
131 215 22 134 217 171 129 222 130 130 233 228
165 250 22 133 251 205 164 0 164 165 12 6
239 54 144 2 146 148 48 26 151 232 90 23
255 70 160 18 162 164 64 42 167 248 106 39
48 185 159 208 211 213 113 91 217 41 155 73
158 171 138 222 201 76 218 73 133 204 254 134

k2next
208 30 138 156 250 126 11 122 182 254 47 168

```

Figure 4.25 Gossamer MixBits function, k1next and k2next creation (Master).

Stage 4: n3, k1next and k2next creation (Slave side)

$$n3 = MIXBITS(n1, n2)$$

$$K1^* = ROT(ROT(n2 + K1 + \pi + n3, n2) + K2 \oplus n3, n1) \oplus n3$$

$$K2^* = ROT(ROT(n1 + K2 + \pi + n3, n1) + K1 + n3, n2) + n3$$

```

COM11 - PuTTY
MixBits
49      114      255      190      49      49      49      49      49      49      49      34
49      114      255      190      49      49      49      49      49      49      49      34
99      182      246      102      185      139      97      190      98      98      201      196
115     199      6       118      201      155      113      206      114      114      217      212
150     234      6       118      236      190      148      241      149      149      252      247
101     244      167      140      172      175      231      188      183      80      51      183
134     20      199      172      204      208      7       220      215      112      83      215
183     102      56       18       253      225      54      237      230      65      98      245
247     132      219      183      153      5       139      214      221      152      224      75
k1next
198     246      36       9       168      52      186      231      236      169      209      105
49      114      255      190      49      49      49      49      49      49      49      34
99      182      246      102      185      139      97      190      98      98      201      196
131     215      22       134      217      171      129      222      130      130      233      228
165     250      22       133      251      205      164      0       164      165      12       6
239     54       144      2       146      148      48      26      151      232      90      23
255     70       160      18       162      164      64      42      167      248      106      39
48      185      159      208      211      213      113      91      217      41      155      73
158     171      138      222      201      76      218      73      133      204      254      134
k2next
208     30       138      156      250      126      11      122      182      254      47      168

```

Figure 4.26 Gossamer MixBits function, k1next and k2next creation (Slave).

Stage 5: Message C creation (Master Side)

$$n1^* = MIXBITS(n3, n2)$$

$$C = ROT(ROT(n3 + K1^* + \pi + n1^*, n3) + K2^* \oplus n1^*, n2) \oplus n1^*$$

```

COM10 - PuTTY
n1'
9       194      235      49       244      244      244      244      244      244      240      34
9       194      235      49       244      244      244      244      244      244      240      34
60      6       225      218      125      79      37      130      38      38      136      196
2       253      5       228      37      131      224      106      18      208      90      45
52      112      5       162      86      181      17      155      68      1      139      79
90      212      70      109      16       6       45      60      209      192      22      137
42      242      209      10       10      132      56      183      136      190      70      49
35      48       58      59      254      112      204      67      124      74      182      19
243     134      98      27      226      85      176      153      25      129      209      223
C
250     68      137      42      22      161      68      109      237      117      33      253

```

Figure 4.27 Gossamer message C creation (Master).

Stage 6: Message C creation (Slave Side)

$$n1^* = MIXBITS(n3, n2)$$

$$C = ROT(ROT(n3 + K1^* + \pi + n1^*, n3) + K2^* \oplus n1^*, n2) \oplus n1^*$$

Row	Col 1	Col 2	Col 3	Col 4	Col 5	Col 6	Col 7	Col 8	Col 9	Col 10	Col 11
n1*											
9	194	235	49	244	244	244	244	244	244	240	34
9	194	235	49	244	244	244	244	244	244	240	34
60	6	225	218	125	79	37	130	38	38	136	196
2	253	5	228	37	131	224	106	18	208	90	45
52	112	5	162	86	181	17	155	68	1	139	79
90	212	70	109	16	6	45	60	209	192	22	137
42	242	209	10	10	132	56	183	136	190	70	49
35	48	58	59	254	112	204	67	124	74	182	19
243	134	98	27	226	85	176	153	25	129	209	223
C											
250	68	137	42	22	161	68	109	237	117	33	253

Figure 4.28 Gossamer message C creation (Slave).

Stage 7: Message D creation (Master side)

$$D = ROT(ROT(n2 + K2^* + ID + n1^*, n2) + K1^* + n1^*, n3) + n1^*$$

Row	Col 1	Col 2	Col 3	Col 4	Col 5	Col 6	Col 7	Col 8	Col 9	Col 10	Col 11
RX											
9	194	235	49	244	244	244	244	244	244	240	34
78	7	47	118	57	57	57	57	57	57	52	102
30	37	186	19	51	183	68	179	240	55	100	14
65	72	186	19	86	218	103	215	19	90	135	49
182	211	62	184	154	212	57	138	10	69	208	154
125	201	98	194	67	8	244	113	246	239	162	3
135	140	77	244	55	253	233	102	235	228	146	37
223	247	165	155	175	146	72	150	30	49	55	208
message D											
233	186	144	205	164	135	61	139	19	38	39	242

Figure 4.29 Gossamer message D creation (Master).

Stage 8: Message D creation (Slave side)

$$D = ROT(ROT(n2 + K2^* + ID + n1^*, n2) + K1^* + n1^*, n3) + n1^*$$

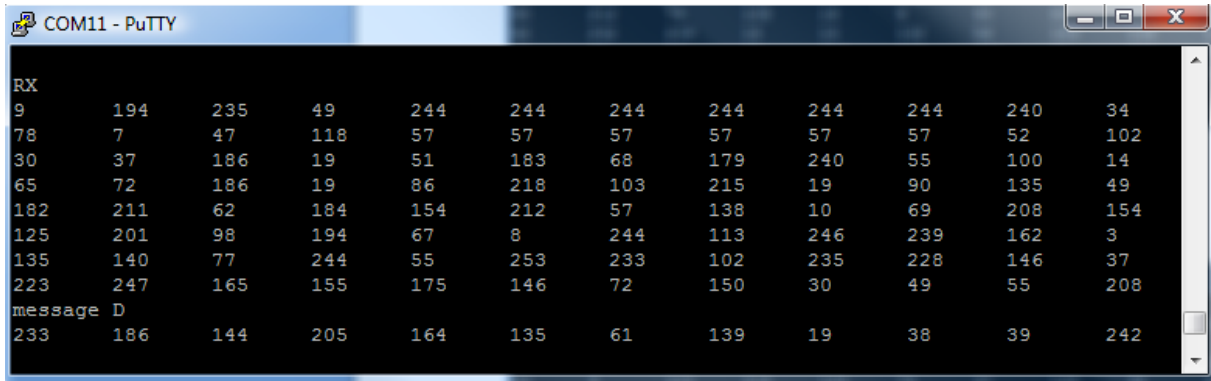


Figure 4.30 Gossamer message D creation (Slave).

Stage 9: IDS, k1 and k2 updating (Master side)

$$n2^* = MIXBITS(n1^*, n3)$$

$$IDS = ROT(ROT(n1^* + K1^* + IDS + n2^*, n1^*) + K2^* \oplus n2^*, n3) \oplus n2^*$$

$$K1 = ROT(ROT(n3 + K2^* + \pi + n2^*, n3) + K1^* + n2^*, n1^*) + n2^*$$

$$K2 = ROT(ROT(IDS + K2^* + \pi + K1, IDS) + K1^* + K1, n2^*) + K1$$

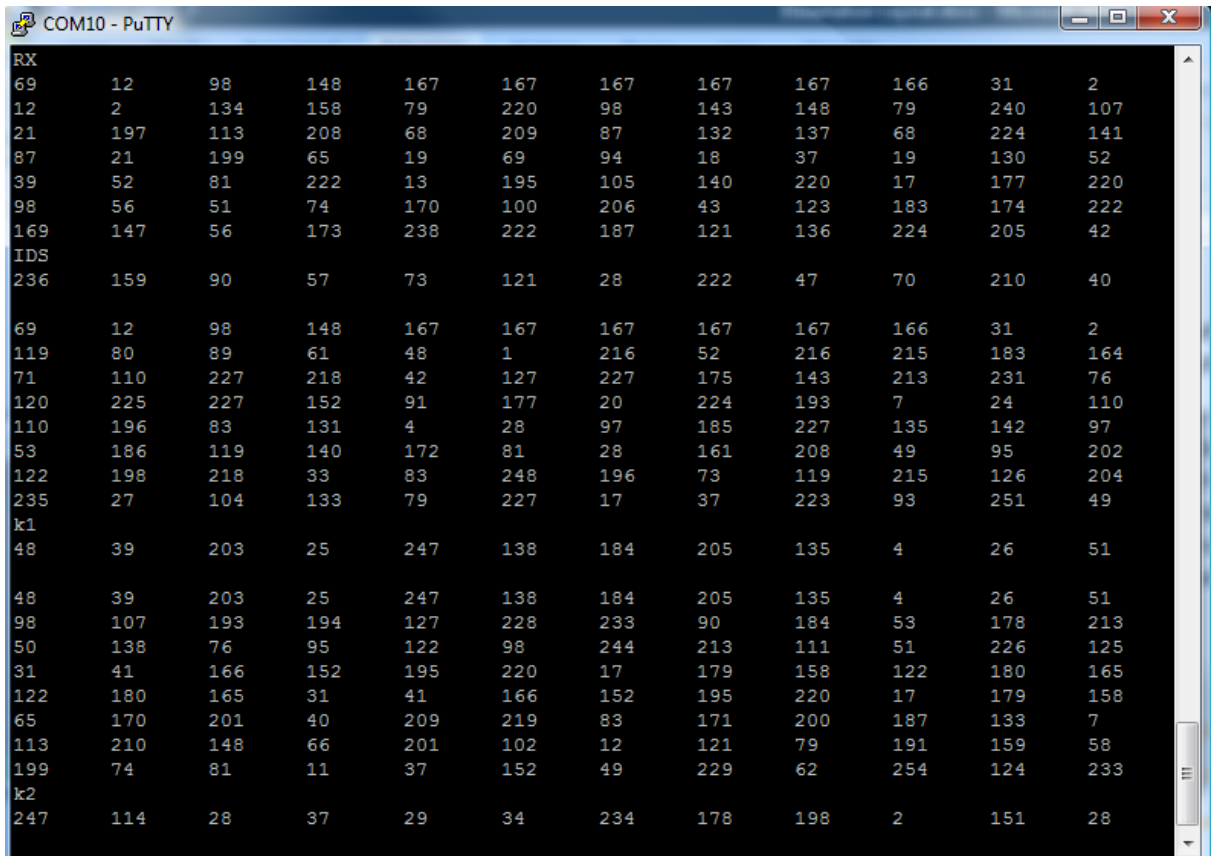


Figure 4.31 Gossamer keys and IDS updating phase (Master).

Stage 10: IDS, k1 and k2 updating (Slave side)

$$n2^* = MIXBITS(n1^*, n3)$$

$$IDS_{old} = IDS$$

$$IDS_{next} = ROT(ROT(n1^* + K1^* + IDS + n2^*, n1^*) + K2^* \oplus n2^*, n3) \oplus n2^*$$

$$K1_{old} = K1$$

$$K1_{next} = ROT(ROT(n3 + K2^* + \pi + n2^*, n3) + K1^* + n2^*, n1^*) + n2^*$$

$$K2_{old} = K2$$

$$K2_{next} = ROT(ROT(IDS_{next} + K2^* + \pi + K1_{next}, IDS_{next}) + K1^* + K1_{next}, n2^*) + K1_{next}$$

```

COM11 - PuTTY
69 12 98 148 167 167 167 167 167 166 31 2
12 2 134 158 79 220 98 143 148 79 240 107
21 197 113 208 68 209 87 132 137 68 224 141
87 21 199 65 19 69 94 18 37 19 130 52
39 52 81 222 13 195 105 140 220 17 177 220
98 56 51 74 170 100 206 43 123 183 174 222
169 147 56 173 238 222 187 121 136 224 205 42
nIDS
236 159 90 57 73 121 28 222 47 70 210 40

69 12 98 148 167 167 167 167 167 166 31 2
119 80 89 61 48 1 216 52 216 215 183 164
71 110 227 218 42 127 227 175 143 213 231 76
120 225 227 152 91 177 20 224 193 7 24 110
110 196 83 131 4 28 97 185 227 135 142 97
53 186 119 140 172 81 28 161 208 49 95 202
122 198 218 33 83 248 196 73 119 215 126 204
235 27 104 133 79 227 17 37 223 93 251 49
k1
48 39 203 25 247 138 184 205 135 4 26 51

48 39 203 25 247 138 184 205 135 4 26 51
98 107 193 194 127 228 233 90 184 53 178 213
50 138 76 95 122 98 244 213 111 51 226 125
31 41 166 152 195 220 17 179 158 122 180 165
122 180 165 31 41 166 152 195 220 17 179 158
65 170 201 40 209 219 83 171 200 187 133 7
113 210 148 66 201 102 12 121 79 191 159 58
199 74 81 11 37 152 49 229 62 254 124 233
k2
247 114 28 37 29 34 234 178 198 2 151 28
  
```

Figure 4.32 Gossamer keys and IDS updating phase (Master).

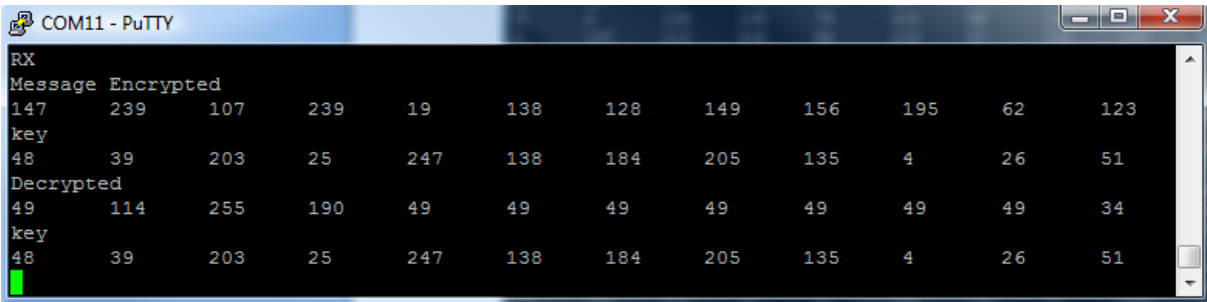
Stage 11: SEA Encryption using k1 (Master side)

```

COM10 - PuTTY
Message
49 114 255 190 49 49 49 49 49 49 34
key
48 39 203 25 247 138 184 205 135 4 26 51
Encrypted
147 239 107 239 19 138 128 149 156 195 62 123
key
48 39 203 25 247 138 184 205 135 4 26 51
  
```

Figure 4.33 SEA encryption (Master)

Stage 12: SEA Decryption using k1 (Slave side)



```
COM11 - PuTTY
RX
Message Encrypted
147 239 107 239 19 138 128 149 156 195 62 123
key
48 39 203 25 247 138 184 205 135 4 26 51
Decrypted
49 114 255 190 49 49 49 49 49 49 49 34
key
48 39 203 25 247 138 184 205 135 4 26 51
```

Figure 4.34 SEA decryption (Slave)

4.5.3 Long-term test

Test Procedure

The goal of this test is to verify the proper functioning of the Gossamer Authentication Protocol and the SEA encryption/decryption algorithm using multiple values and multiple rounds.

The test-Master and the test-Slave are pre-configured to loop indefinitely executing the following operations:

- Both devices: mutual authentication between the test-Slave and the test-Master
- Both devices: updating values for the next round.
- Master: encrypting a 12-byte message using the SEA encryption algorithm (using the Gossamer key k1) and transmitting the payload to the test-Slave.
- Slave: receiving the payload from the test-Master and decrypting it using the SEA decryption algorithm and the Gossamer key k1.

The test-Master uses a delay function before transmitting messages over the radio to allow for better readability of the UART output. Both the test-Slave and the test-Master output informational messages to the UART during each loop iteration.

The time to complete an iteration of the main loop in both programs was estimated at approximately 1.5 seconds. The test-Master and the test-Slave programs were left running for 7 days. It is estimated that both programs will execute approximately 403200 authentication and encryption/decryption rounds.

Test data

Master initial values:

```
unsigned char idata Pi[12] = { 0x32, 0x43, 0xF6, 0xA8, 0x88, 0x5A, 0x30,
0x8D, 0x31, 0x31, 0x98, 0xA2 };
```

```

unsigned char idata IDS[12] = { 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01,
0x01, 0x01, 0x01, 0x01, 0x01 };
unsigned char idata ID[12] = { 0x44, 0x44, 0x44, 0x44, 0x44, 0x44, 0x44,
0x44, 0x44, 0x44, 0x44, 0x44 };
unsigned char idata k1[12] = { 0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10,
0x10, 0x10, 0x10, 0x10, 0x10 };
unsigned char idata k2[12] = { 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
0x20, 0x20, 0x20, 0x20, 0x20 };
unsigned char idata n1[12] = { 0x22, 0x22, 0x22, 0x22, 0x22, 0x22, 0x22,
0x22, 0x22, 0x22, 0x22, 0x22 };
unsigned char idata n2[12] = { 0x23, 0x23, 0x23, 0x23, 0x23, 0x23, 0x23,
0x23, 0x23, 0x23, 0x23, 0x23 };

```

Figure 4.35 Code: Master Initial Values

Slave initial values:

```

unsigned char idata Pi[12] = { 0x32, 0x43, 0xF6, 0xA8, 0x88, 0x5A, 0x30,
0x8D, 0x31, 0x31, 0x98, 0xA2 };
unsigned char idata IDS[12] = { 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01,
0x01, 0x01, 0x01, 0x01, 0x01 };
unsigned char idata ID[12] = { 0x44, 0x44, 0x44, 0x44, 0x44, 0x44, 0x44,
0x44, 0x44, 0x44, 0x44, 0x44 };
unsigned char idata k1[12] = { 0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10,
0x10, 0x10, 0x10, 0x10, 0x10 };
unsigned char idata k2[12] = { 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
0x20, 0x20, 0x20, 0x20, 0x20 };

```

Figure 4.36 Code: Slave Initial Values

After the Gossamer round the Master will use the modified key k1 to encrypt a message (temp array) and send to the Slave. After successful transmission the Slave will use modified k1 to decrypt the message and display it.

Test Results

Both the test-Master and the test-Slave were running continuously for 6 days and 23 hours and successfully executed approximately 400 000 mutual authentications and encryption/decryption rounds.

The UART output was periodically monitored and no abnormalities were discovered.

5. Performance Analysis

5.1 Memory Code Space Requirements on nRF9E5

The Master and the Slave prototypes used in a Long-term test required the following code space:

Master: 3923 bytes + 80 bytes of xdata.

Slave: 3937 bytes + 80 bytes of xdata.

The main Gossamer Protocol function loop uses UART to intermittently output the results throughout the round. In order to verify the real size of the Gossamer loop without any UART overhead a special version of the program was compiled without any calls to the PutString or printArray functions. The results after these changes will be shown in a form of an extract from the Raisonance LX51 Linker Map File which can be found in Appendix A.

The RAM storage requirements of the Master (229 bytes) and the Slave (244 bytes) seem high but it has to be noted that all arrays holding 96-bit numbers used by the Gossamer Protocol are initialized and stored in idata memory during the runtime of the program. In consequence 210 bytes of idata memory is used by the Master and the Slave. Approximately 80% of this space can be saved by moving the 96-bit values to the EEPROM trading off code space required by the external memory read/write functions.

The total code space requirement by all the Gossamer-related functions is estimated at 1647bytes (66F Hex) on the Master side and 1710 bytes (6AE Hex) on the Slave side.

The SEA functions code space requirements are identical on both the Master and the Slave programs and equal to 589 bytes (24D Hex).

It has to be noted that the Raisonance RC51 compiler imports a LIB51 library which requires 552 bytes (228 Hex). This library is shared by many functions performing mathematical operations and is automatically imported even if only the SEA functions were to be implemented. In consequence, the code space requirements of the LIB51 have to be taken under consideration when estimating the total requirements.

5.2 Execution Speed

The execution speed of different parts of the code was analyzed using an nRF9E5 timer interrupt set to 1 millisecond ticks and small timer handling functions. The timer was reset before entering a given block of code and the timer value was collected at the exit of the block.

SEA Encryption/Decryption

The full SEA (96, 8) encryption and decryption of a 12-byte block using 12-byte key and 93 rounds takes 27 milliseconds on an nRF9E5 microcontroller running at 16MHz. This gives an encryption/decryption throughput of 705 bytes per second.

Gossamer Authentication

The full round of the Gossamer Protocol in the prototype program with no UART output (all PutString function calls removed) took 984 milliseconds on the Master side and 988 milliseconds on the Slave side. Both devices used the simplified radio protocol described in section 4.3. It has to be noted that the Master uses a longDelay function which loops for 280ms before each transmission (TX) attempt. At this time the Slave loops in the receiving mode (RX) waiting for messages. There are 3 TX attempts (messages A, B and C) so the total of $3 \times 280\text{ms}$ can be subtracted from the total loop time on both the Master and the Slave side.

The full Gossamer loop time without the TX delay function:

- Master: $984\text{ms} - 3 \times 280\text{ms} = 144\text{ms}$
- Slave: $988\text{ms} - 3 \times 280\text{ms} = 148\text{ms}$

The Gossamer Protocol speed was also analyzed per major protocol stages:

- Message A and B creation: 2ms each (Master).
- Message C creation: 65ms (Master and Slave).
- Number n1 and n2 extraction: 2ms each (Slave).
- Message D creation and verification: 3ms (Master).
- Message D creation 2ms: (Slave).
- Keys and IDS update: 38ms (Master and Slave).

6. Conclusions and Recommendations

6.1 Conclusions

Lightweight Authentication and Encryption protocols have emerged to fill the security void created by the transition from desktop to mobile environments. Fast processing and large memory has characterised desktop technologies. By contrast, mobile technologies are characterised by their small processing power and small memory. Authentication and encryption protocols designed for desktop technologies cannot be easily ported to mobile Resource Limited Devices (RLDs).

The central theme of this dissertation is that lightweight authentication and encryption protocols can fulfil the requirements of secure communications between RLDs without hardware modification. An augmentation of the Gossamer authentication protocol that incorporates elements of the Scalable Encryption Algorithm (SEA) was implemented to confirm this assertion. Cora Data's wireless sensor development board, comprising the Nordic Semiconductor nRF9E5 microcontroller and auxiliary radio communications circuitry was used as the reference platform. The implementation, in software, demonstrates successful accomplishment of the key objectives of secure communications, but at a cost. Success has been achieved by greatly simplifying the radio protocol and using almost the entire code space of 4 Kilobytes allowed by the nRF9E5 microcontroller for the implementation of the security mechanisms. As a consequence, there is zero code space left for the other tasks involved in the normal operation of an Infrastructure Wireless Sensor Network, such as sampling the ADC convertor and forming a data payload with the results.

The research objectives, outlined in section 1.3, have been fulfilled. A literature review comprising an overview of the security issues with respect to RLDs and their limitations (section 3), an analysis of authentication (section 2.2) and encryption (section 2.3) has been completed and has established that the Gossamer and SEA protocols, are the most suitable of the family of ultra-lightweight security protocols for implementation on RLDs. The algorithms are current, resistant to attacks and cryptanalysis and their design has been focused on providing solutions for resource limited devices. In addition, they can be implemented on an 8 bit platform. An augmented Gossamer protocol that incorporates elements of the SEA is presented as a possible solution to the implementation of security in networks of RLDs.

A major goal of this dissertation is to examine code space requirements of the augmented protocol's implementation (since memory is a critical resource). The target is to provide secure communications with protocols that subsume as little of the memory as possible of the RLD. Although Gossamer uses basic mathematical operations, which are easy to implement in hardware, the software implementation on an 8-bit CPU involves a great deal of code space overhead. The performance analysis (section 5.1) shows that the total code space required by the Gossamer functions (~1700 Bytes) including the necessary RC51 libraries (552 Bytes) can be

estimated at approximately 2200 Bytes, which is 55% of the code space available on the reference platform. The overhead mainly relates to operations on large numbers that have to be split into arrays with elements equal to the word size of the CPU. This leaves little room for the implementation of the radio protocol (hence the need for simplification) and zero room for ADC or other functionality.

Additionally, the execution speed for each full round of the Gossamer Protocol (144 – 148ms) is relatively high (reflecting the limitations of nRF9E5 processing power). The simplicity of the underlying mathematical calculations would imply fast performance. In fact the actual performance varies significantly from that expected. This may have adverse consequences on the efficiency of the communications protocol. Further code optimisation and/or native assembly code would reduce code space requirement and improve performance, but not by a magnitude large enough to justify the implementation of a software implementation of the Gossamer protocol on the reference platform. However, if another microcontroller without so strict memory limitations is used and the performance is regarded as satisfactory then the mechanism proposed can be considered for implementation.

The SEA (96,8) implementation results were much more promising than the Gossamer ones. As expected from an algorithm designed to be adapted easily to the native word size of the CPU, the code space footprint is very small (589 Bytes). Even when the RC51 libraries overhead is taken into consideration (552 Bytes), the total size of 1141 Bytes is just below 28% of the total code space available on the nRF9E5. SEA has not been proven to be insecure to date, thus it can be recommended for microcontroller implementations with associated low data throughput requirements.

The code space requirement to implement Gossamer combined with the code space required by SEA is 3341 Bytes (2200 Bytes + 1141 Bytes) or 83% of available code space. The remainder of the code space is subsumed by simple radio functionality.

Given the associated memory limitations, lack of hardware support for cryptographic primitives and the difficulty of implementing code banking with any degree of performance efficiency, the nRF9E5 cannot be recommended as a suitable platform on which to implement native authentication and encryption in security demanding wireless sensor networks. Low cost microcontroller alternatives, such as the Texas Instruments CC430 family of microcontrollers with an embedded UHF radio transceiver and hardware support for 128-bit AES encryption may be viable.

6.2 Recommendations for future work

The promising results of the SEA (96, 8) algorithm implementation (with respect to code size and no. of cycles required to complete the protocol) would suggest that there is room for further investigation in relation to key size and the associated security that this brings. It would be interesting to implement a (192, 8) version using

a 24-byte key and block size. A comparative framework could then be drawn up to assess performance of both implementations.

In consequence of the significantly high code space overhead required by the software implementation of Gossamer, further study of authentication and the authentication protocols needs to emerge. The need for authentication protocols that can be implemented in terse code and negate all aspects of security breach remains a priority in the field of wireless sensor networks. There are additional implications for power consumption, battery life, signal strength and propagation distance that will have an influence on the evolution of both sensors and security protocols.

Implementation of the prototype on a larger scale (multiple sensors, single master and the back-end server) may significantly affect performance. Further research in this respect would identify performance-related issues and further test the suitability of the proposed solution for Infrastructure Wireless Sensor Networks.

Additionally, an approach that combines authentication, encryption and key exchange in a single protocol with shared keys of identical length may prove to be a useful line of academic enquiry.

References

- Abramson, N., 1970. The aloha system: Another alternative for computer communications. In *Proceedings of the November 17-19, 1970, fall joint computer conference*. pp. 281–285.
- Ahmed, E.G., Shaaban, E. & Hashem, M., 2010. Lightweight Mutual Authentication Protocol for Low Cost RFID Tags. *International Journal of Network Security & Its Application (IJNSA), Academy & Industry Research Collaboration Center (AIRCC)*.
- Akyildiz, I.F., Su, W., Sankarasubramaniam, Y. & Cayirci, E., 2002. Wireless sensor networks: a survey. *Computer networks*, 38(4), 393–422.
- Andem, V.R., 2003. *A cryptanalysis of the tiny encryption algorithm*. Citeseer.
- ARM Ltd., 2009a. Keil C51 Compiler Basics. Available at: <http://www.esacademy.com/automation/docs/c51primer/c02.htm> [Accessed May 18, 2009].
- ARM Ltd., 2009b. LX51 User's Guide: Code Banking. Available at: http://www.keil.com/support/man/docs/lx51/lx51_codebanking.htm [Accessed January 6, 2010].
- Bárász, M., Boros, B., Ligeti, P., Lója, K. & Nagy, D., 2007a. Breaking LMAP. *Proc. of RFIDSec*, 7. Available at: <http://www.cs.elte.hu/~turul/pubs/lmap.pdf>.
- Bárász, M., Boros, B., Ligeti, P., Lója, K. & Nagy, D., 2007b. Passive attack against the M2AP mutual authentication protocol for RFID tags. In *Proc. of First International EURASIP Workshop on RFID Technology*. Available at: <http://www.cs.elte.hu/~turul/pubs/mmap.pdf>.
- Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J., Seurin, Y. & Vikkelsoe, C., 2007. PRESENT: An ultra-lightweight block cipher. *Lecture Notes in Computer Science*, 4727, 450.
- Cao, T., Bertino, E. & Lei, H., 2009. Security Analysis of the SASI Protocol. *IEEE Transactions on Dependable and Secure Computing*, 73–77.
- Chien, H., 2007. SASI: A New Ultralightweight RFID Authentication Protocol Providing Strong Authentication and Strong Integrity. *IEEE Transactions on Dependable and Secure Computing*, 4(4), 337-340.
- Chien, H. & Huang, C.W., 2007. Security of ultra-lightweight RFID authentication protocols and its improvements. *ACM SIGOPS Operating Systems Review*, 41(4), 86.
- D'Arco, P. & De Santis, A., 2008. *From Weaknesses to Secret Disclosure in a Recent Ultra-Lightweight RFID Authentication Protocol*, Cryptology ePrint Archive. <http://eprint.iacr.org/2008/470>, 2008. Available at: <http://eprint.iacr.org/2008/470.pdf>.

- Daemen, J. & Rijmen, V., 1999. AES proposal: Rijndael.
- Eastlake, D. & Jones, P., 2001. *US secure hash algorithm 1 (SHA1)*, RFC 3174, September 2001.
- Eisenbarth, T., Kumar, S., Paar, C., Poschmann, A. & Uhsadel, L., 2007. A survey of lightweight-cryptography implementations. *IEEE Design & Test of Computers*, 522–533.
- EPCGlobal, 2008. EPCglobal UHF Class 1 Gen 2. Available at: <http://www.epcglobalinc.org/standards/uhfclg2> [Accessed August 10, 2009].
- Federal Information Processing Standards, 1993. FIPS 46-2 - (DES), Data Encryption Standard. Available at: <http://www.itl.nist.gov/fipspubs/fip46-2.htm> [Accessed October 25, 2009].
- Hartung, C., Balasalle, J. & Han, R., 2005. Node compromise in sensor networks: The need for secure systems. *Department of Computer Science University of Colorado at Boulder*. Available at: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.134.8146&rep=rep1&type=pdf>.
- Hegazy, A.E., Darwish, A.M. & El-Fouly, R., 2007. Reducing μ TESLA memory requirements.
- Hernandez-Castro, J.C., Estevez-Tapiador, J.M., Ribagorda-Garnacho, A. & Ramos-Alvarez, B., 2006. Wheedham: An automatically designed block cipher by means of genetic programming. In *Proc. of CEC*. pp. 192–199.
- Hernandez-Castro, J.C., Tapiador, J.M., Peris-Lopez, P. & Quisquater, J.J., 2008. Cryptanalysis of the SASI Ultralightweight RFID Authentication Protocol with Modular Rotations. *Arxiv preprint arXiv:0811.4257*.
- HINT Project, 2010. *Research Project: HINT Project*. Letterkenny Institute of Technology.
- Hong, S., Hong, D., Ko, Y., Chang, D., Lee, W. & Lee, S., 2004. Differential Cryptanalysis of TEA and XTEA. *Information Security and Cryptology-ICISC 2003*, 402–417.
- Jinwala, D.C., Patel, D.R. & Dasgupta, K.S., 2008. Investigating and Analyzing the Lightweight ciphers for Wireless Sensor Networks.
- Juels, A., 2005. Strengthening EPC tags against cloning. In *Proceedings of the 4th ACM workshop on Wireless security*. p. 76. Available at: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.68.6553&rep=rep1&type=pdf>.
- Juels, A., 2006. RFID security and privacy: A research survey. *IEEE Journal on Selected Areas in Communications*, 24(2), 381–394.

- Kamble, P., Kshirsagar, R.V. & Mankar, K., 2007. Wireless Sensor Network Architecture. Available at: http://www.ieee-spce.org/colloquium/proceedings/Communication_and_Networking/spit-1.pdf.
- Karlof, C., Sastry, N. & Wagner, D., 2004. TinySec: a link layer security architecture for wireless sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*. pp. 162–175. Available at: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.61.4930&rep=rep1&type=pdf>.
- Kelsey, J., Schneier, B. & Wagner, D., 1997. Related-key cryptanalysis of 3-way, biham-des, cast, des-x, newdes, rc2, and tea. *Information and Communications Security*, 233–246.
- Klimov, A. & Shamir, A., 2004. Cryptographic Applications of T-functions. *Lecture notes in computer science*, 248–261.
- Ko, Y., Hong, S., Lee, W., Lee, S. & Kang, J.S., 2004. Related key differential attacks on 27 rounds of XTEA and full-round GOST. In *Fast Software Encryption*. pp. 299–316.
- Lee, Y.C., Hsieh, Y.C., You, P.S. & Chen, T.C., 2009. A New Ultralightweight RFID Protocol with Mutual Authentication. In *Information Engineering, 2009. ICIE'09. WASE International Conference on*. pp. 58–61.
- Leong, K.S., NG, M.L. & Engels, D.W., 2006. EPC Network Architecture. *Auto-ID Labs: EPC Network Architecture*. Available at: <http://www.autoidlabs.org/uploads/media/AUTOIDLABS-WP-SWNET-012.pdf> [Accessed November 26, 2009].
- Li, T. & Deng, R., 2007. Vulnerability analysis of EMAP-an efficient RFID mutual authentication protocol. *Proc. of AReS*, 7. Available at: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.63.6430&rep=rep1&type=pdf>.
- Li, T. & Wang, G., 2007. Security analysis of two ultra-lightweight RFID authentication protocols. *INTERNATIONAL FEDERATION FOR INFORMATION PROCESSING-PUBLICATIONS-IFIP*, 232, 109.
- Liu, D. & Ning, P., 2004. Multilevel μ TESLA: Broadcast authentication for distributed sensor networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(4), 800–836.
- Lu, J., 2009. Related-key rectangle attack on 36 rounds of the XTEA block cipher. *International Journal of Information Security*, 8(1), 1–11.
- Menezes, A.J., Oorschot, P.C.V. & Vanstone, S.A., 1997. *Handbook of applied cryptography*, CRC Press.
- Mollin, R.A., 2007. *An introduction to cryptography*, CRC Press.
- Moon, D., Hwang, K., Lee, W., Lee, S. & Lim, J., 2002. Impossible differential cryptanalysis

- of reduced round XTEA and TEA. In *Fast Software Encryption*. pp. 117–121.
- Needham, R.M. & Wheeler, D.J., 1997. *eXtended Tiny Encryption Algorithm*, October.
- Ni, S.Y., Tseng, Y.C., Chen, Y.S. & Sheu, J.P., 1999. The broadcast storm problem in a mobile ad hoc network. In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*. p. 162. Available at: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.123.5000&rep=rep1&type=pdf>.
- Nordic Semiconductors, 2009a. NORDIC SEMICONDUCTOR - nRF905 Multiband Transceiver. Available at: <http://www.nordicsemi.com/index.cfm?obj=product&act=display&pro=83> [Accessed January 5, 2010].
- Nordic Semiconductors, 2009b. NORDIC SEMICONDUCTOR - nRF9E5 Multiband Transceiver/MCU/ADC. Available at: <http://www.nordicsemi.com/index.cfm?obj=product&act=display&pro=82> [Accessed November 8, 2009].
- Ouafi, K. & Vaudenay, S., 2009. Smashing SQUASH-0. In *Advances in Cryptology - EUROCRYPT 2009*. pp. 300-312. Available at: http://dx.doi.org/10.1007/978-3-642-01001-9_17 [Accessed January 6, 2010].
- Peris-Lopez, P., Hernandez-Castro, J., Tapiador, J. & Ribagorda, A., 2009. Advances in Ultralightweight Cryptography for Low-Cost RFID Tags: Gossamer Protocol. In *Information Security Applications*. pp. 56–68.
- Peris-Lopez, P., Hernandez-Castro, J.C., Estevez-Tapiador, J.M. & Ribagorda, A., 2006a. EMAP: An efficient mutual-authentication protocol for low-cost RFID tags. *LECTURE NOTES IN COMPUTER SCIENCE*, 4277, 352.
- Peris-Lopez, P., Hernandez-Castro, J.C., Estevez-Tapiador, J.M. & Ribagorda, A., 2006b. LMAP: A real lightweight mutual authentication protocol for low-cost RFID tags. In *Workshop on RFID Security*. pp. 12–14. Available at: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.110.2082&rep=rep1&type=pdf>.
- Peris-Lopez, P., Hernandez-Castro, J.C., Estevez-Tapiador, J.M. & Ribagorda, A., 2006c. M²AP: A Minimalist Mutual-Authentication Protocol for Low-Cost RFID Tags. *Lecture Notes in Computer Science*, 4159, 912.
- Peris-Lopez, P., Hernandez-Castro, J.C., Tapiador, J.M., van der Lubbe, J.C., Singh, M.K., Liang, G., Vaidya, N., Shanmugapriya, D., Padmavathi, G. & Kish, L.L., 2009. Security Flaws in a Recent Ultralightweight RFID Protocol. *Arxiv preprint arXiv:0910.2115*.
- Perrig, A., Canetti, R., Song, D. & Tygar, J.D., 2001. Efficient and secure source authentication for multicast. In *Network and Distributed System Security Symposium, NDSS*. pp. 35–46. Available at:

- <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.18.1680&rep=rep1&type=pdf>.
- Perrig, A., Szewczyk, R., Tygar, J.D., Wen, V. & Culler, D.E., 2002. SPINS: Security protocols for sensor networks. *Wireless networks*, 8(5), 521–534.
- Poschmann, A., Leander, G., Schramm, K. & Paar, C., 2007. New light-weight crypto algorithms for RFID. In *Proceedings of The IEEE International Symposium on Circuits and Systems*. pp. 1843–1846. Available at: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.80.1217&rep=rep1&type=pdf>.
- Rabin, M.O., 1979. Digitalized signatures and public-key functions as intractable as factorization. *MtT/LCS/TR-212*.
- Raisonance SAS, 2010. Raisonance, Corporate home page. Available at: <http://www.raisonance.com/> [Accessed June 1, 2010].
- Ranasinghe, D.C. & Cole, P.H., 2008. *Networked RFID Systems and Lightweight Cryptography*, Springer Berlin Heidelberg. Available at: <http://dx.doi.org/10.1007/978-3-540-71641-9> [Accessed December 1, 2008].
- Rinne, S., Eisenbarth, T. & Paar, C., 2007. Performance analysis of contemporary light-weight block ciphers on 8-bit microcontrollers. *ECRYPT*, 33.
- Rivest, R.L., 1995. The RC5 encryption algorithm. *Dr Dobb's Journal-Software Tools for the Professional Programmer*, 20(1), 146–149.
- el Ruptor, M., 2007. File:XXTEA.png - Wikipedia, the free encyclopedia. Available at: <http://en.wikipedia.org/wiki/File:XXTEA.png> [Accessed September 19, 2010].
- Russell, M.D., 2004. Tinyness: an overview of TEA and related ciphers. *Draft v0.3*, 3.
- Saarinen, M.J., 1998. *Cryptanalysis of Block Tea*, unpublished manuscript.
- Sarma, S.E., 2001. *Towards the five-cent tag*, Technical Report MIT-AUTOID-WH-006, MIT Auto ID Center, 2001. Available at: <http://www.autoidlabs.org/uploads/media/mit-autoid-wh-006.pdf>.
- Sarma, S.E., Weis, S.A. & Engels, D.W., 2003. RFID systems and security and privacy implications. *Lecture notes in computer science*, 454–469.
- Schneier, B., 1996. *Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition* 2nd ed., Wiley.
- Shamir, A., 2008. SQUASH – A New MAC with Provable Security Properties for Highly Constrained Devices Such as RFID Tags. In *Fast Software Encryption*. pp. 144-157. Available at: http://dx.doi.org/10.1007/978-3-540-71039-4_9 [Accessed January 6, 2010].

- Standaert, F., Piret, G., Gershenfeld, N. & Quisquater, J., 2006. SEA: A scalable encryption algorithm for small embedded applications. *Lecture Notes in Computer Science*, 3928, 222.
- Sun, H.M., Ting, W.C. & Wang, K.H., 2008. *On the security of chien's ultralightweight RFID authentication protocol*, Cryptology ePrint Archive, Report 2008/083, 2008. Available at: <http://eprint.iacr.org/2008/083.pdf>.
- Vault Information Services, 2009. 8052.com - The Online 8051/8052 Microcontroller Resource - 8052.com. Available at: <http://www.8052.com/> [Accessed April 7, 2009].
- Wheeler, D. & Needham, R., 1994. TEA, a tiny encryption algorithm. In *Fast Software Encryption*. pp. 363–366.
- Wheeler, D. & Needham, R., 1998. *XXTEA: Correction to XTEA*, Technical report, Computer Laboratory, University of Cambridge.
- Wi-Fi Alliance, W.F., 2003. Wi-Fi Protected Access: Strong, standards-based, interoperable security for today's Wi-Fi networks. , 1(2004), 29–03.
- Yarrkov, E., 2010. *Cryptanalysis of XXTEA*, Available at: <http://eprint.iacr.org/2010/254> [Accessed May 28, 2010].
- Ye, W., Heidemann, J. & Estrin, D., 2002. An energy-efficient MAC protocol for wireless sensor networks. In *IEEE INFOCOM*. pp. 1567–1576.
- Yu-Long, S., Qing-Qi, P.E.I. & Jian-Feng, M.A., 2007. microTESLA: Broadcast Authentication Protocol for Multiple-Base-Station Sensor Networks.

Appendix A

Master Linker Map

TYPE	BASE	LENGTH	RELOCATION	SEGMENT NAME
*****		DATA/IDATA	MEMORY	*****
REG	0000H	0008H	ABSOLUTE	"REG BANK 0"
DATA	0008H	0018H	OVERLAID UNIT	_DGROUP02_
	0008H	0002H	-----	?DT?_ChangeRXAddress?MASTER
	0008H	0009H	-----	?DT?_gossamerMaster?MASTER
	0011H	0004H	-----	?DT?_ReceiveMode?MASTER
	0011H	0007H	-----	?DT?_bitRotation?MASTER
	0018H	0003H	-----	?DT?_getModulo96?MASTER
	0018H	0004H	-----	?DT?_indexShift?MASTER
	001CH	0003H	-----	?DT?_arrayReverse?MASTER
	0018H	0006H	-----	?DT?_bitShift?MASTER
	0011H	0002H	-----	?DT?_TransmitBytes?MASTER
	0011H	0003H	-----	?DT?_mixBits?MASTER
	0014H	0001H	-----	?DT?_additionMod96?MASTER
	0011H	0004H	-----	?DT?_sea?MASTER
	0015H	0009H	-----	?DT?_seaCryptRound?MASTER
	001EH	0001H	-----	?DT?_seaWordRotation?MASTER
	0015H	0008H	-----	?DT?_seaKeyRound?MASTER
	0011H	0007H	-----	?DT?_printArray?MASTER
	0018H	0008H	-----	?DT?_itoa?MASTER
DATA	0020H	0002H	OVERLAID UNIT	_DGROUP01_
	0020H	0002H	-----	?DT?_subtractionMod96?MASTER
IDATA	0022H	00C3H	OVERLAID UNIT	_IGROUP02_
	0022H	00B4H	-----	?ID?_gossamerMaster?MASTER
	00D6H	000CH	-----	?ID?_mixBits?MASTER
	00D6H	000FH	-----	?ID?_sea?MASTER
	00D6H	0003H	-----	?ID?_itoa?MASTER
IDATA	00E5H	0001H	*** STACK ***	_STACK
*****		PDATA/XDATA	MEMORY	*****
	0000H	0FB0H		*** GAP ***
XDATA	0FB0H	0048H	OVERLAID UNIT	_XGROUP02_
	0FB0H	0008H	-----	?XD?main?MASTER
	0FB8H	0040H	-----	?XD?gossamerMaster?MASTER
*****		CODE	MEMORY	*****
TYPE	BASE	LENGTH	RELOCATION	SEGMENT NAME
CODE	0000H	0003H	ABSOLUTE	
CODE	0003H	008EH	INBLOCK	?PR?MOVES?LIB51
CODE	0091H	0228H	UNIT	?PR?LIB51
CODE	02B9H	0018H	UNIT	?PR?C51_STARTUP?
CODE	02D1H	000DH	UNIT	?PR?_SpiReadWrite?MASTER
CODE	02DEH	0008H	UNIT	?PR?_PutChar?MASTER
CODE	02E6H	0008H	UNIT	?PR?GetChar?MASTER
CODE	02EEH	0026H	UNIT	?PR?_PutString?MASTER
CODE	0314H	0021H	UNIT	?PR?SetClock?MASTER
CODE	0335H	0020H	UNIT	?PR?InitUartTimer1?MASTER
CODE	0355H	0014H	UNIT	?PR?longDelay?MASTER
CODE	0369H	0032H	UNIT	?PR?_ChangeRXAddress?MASTER
CODE	039BH	0015H	UNIT	?PR?InitRadio?MASTER
CODE	03B0H	004DH	UNIT	?PR?_TransmitBytes?MASTER

CODE	03FDH	006CH	UNIT	?STR?MASTER
CODE	0469H	0056H	UNIT	?PR?_ReceiveMode?MASTER
CODE	04BFH	00B0H	UNIT	?PR?_itoa?MASTER
CODE	056FH	003AH	UNIT	?PR?_printArray?MASTER
CODE	05A9H	004EH	UNIT	?PR?_getModulo96?MASTER
CODE	05F7H	0028H	UNIT	?PR?_arrayReverse?MASTER
CODE	061FH	0044H	UNIT	?PR?_indexShift?MASTER
CODE	0663H	00CFH	UNIT	?PR?_bitShift?MASTER
CODE	0732H	0045H	UNIT	?PR?_bitRotation?MASTER
CODE	0777H	0010H	UNIT	?PR?_xorArrays?MASTER
CODE	0787H	0041H	UNIT	?PR?_additionMod96?MASTER
CODE	07C8H	0058H	UNIT	?PR?_subtractionMod96?MASTER
CODE	0820H	0054H	UNIT	?PR?_mixBits?MASTER
CODE	0874H	004EH	UNIT	?PR?_seaSBOX?MASTER
CODE	08C2H	002FH	UNIT	?PR?_seaBitRotation?MASTER
CODE	08F1H	0036H	UNIT	?PR?_seaWordRotation?MASTER
CODE	0927H	0064H	UNIT	?PR?_seaCryptRound?MASTER
CODE	098BH	0050H	UNIT	?PR?_seaKeyRound?MASTER
CODE	09DBH	00E6H	UNIT	?PR?_sea?MASTER
CODE	0AC1H	000FH	UNIT	?PR?_copyArray?MASTER
CODE	0AD0H	03EDH	UNIT	?PR?gossamerMaster?MASTER
CODE	0EBDH	0028H	UNIT	?PR?main?MASTER

EXECUTABLE SUMMARY:

Total INTERNAL RAM storage requirement:	00E5H (229)
Total EXTERNAL RAM storage requirement:	0048H (72)
Total CODE storage requirement:	0EE5H (3813)

Slave Linker Map

TYPE	BASE	LENGTH	RELOCATION	SEGMENT NAME
*** **		DATA/IDATA	MEMORY	*****
REG	0000H	0008H	ABSOLUTE	"REG BANK 0"
DATA	0008H	0017H	OVERLAID UNIT	_DGROUP02_
	0008H	0002H	-----	?DT?_ChangeRXAddress?SLAVE
	0008H	0009H	-----	?DT?gossamerSlave?SLAVE
	0011H	0004H	-----	?DT?_ReceiveMode?SLAVE
	0011H	0007H	-----	?DT?_bitRotation?SLAVE
	0018H	0003H	-----	?DT?_getModulo96?SLAVE
	0018H	0004H	-----	?DT?_indexShift?SLAVE
	001CH	0003H	-----	?DT?_arrayReverse?SLAVE
	0018H	0006H	-----	?DT?_bitShift?SLAVE
	0011H	0002H	-----	?DT?_subtractionMod96?SLAVE
	0011H	0003H	-----	?DT?_mixBits?SLAVE
	0014H	0001H	-----	?DT?_additionMod96?SLAVE
	0011H	0004H	-----	?DT?_sea?SLAVE
	0015H	0009H	-----	?DT?_seaCryptRound?SLAVE
	001EH	0001H	-----	?DT?_seaWordRotation?SLAVE
	0015H	0008H	-----	?DT?_seaKeyRound?SLAVE
DATA	001FH	000FH	OVERLAID UNIT	_DGROUP01_
	001FH	0007H	-----	?DT?_printArray?SLAVE
	0026H	0008H	-----	?DT?_itoa?SLAVE
IDATA	002EH	00C3H	OVERLAID UNIT	_IGROUP02_
	002EH	00B4H	-----	?ID?gossamerSlave?SLAVE
	00E2H	000CH	-----	?ID?_mixBits?SLAVE
	00E2H	000FH	-----	?ID?_sea?SLAVE

Appendix B

```
/*
Copyright 2010 Piotr Ksiazak
Filename: Master.c
Project : MSc - IWSN Experimental Master
Version 1.0: Initial release
*/

#include <reg9e5.h>
#include <intri51.h>
#define POWER      3           // 0=min power...3 = max power
#define HFREQ      1           // 0=433MHz, 1=868/915MHz
#define CHANNEL    351        // Channel number: f(MHz) =
                               // (422.4+CHANNEL/10)*(1+HFREQ)
#pragma REGPARMS             // pass arguments to registers

// SPI access
unsigned char SpiReadWrite(unsigned char b)
{
    EXIF &= ~0x20;           // Clear SPI interrupt
    SPI_DATA = b;           // Move byte to send to SPI data register
    while((EXIF & 0x20) == 0x00) // Wait until SPI hs finished transmitting
        ;
    return SPI_DATA;
}

// Send character to UART
void PutChar(char c)
{
    while(!TI)
        ;
    TI = 0;
    SBUF = c;
}

// Read character from UART
unsigned char GetChar(void)
{
    while(!RI)
        ;
    RI = 0;
    return SBUF;
}

// Send string to UART
void PutString(const char *s)
{
    while(*s != 0)
        PutChar(*s++);
}

// Switch to 16MHz clock:
void SetClock(void)
{
    unsigned char cklf;

    RACSN = 0; // Set CSN on the radio to low (Radio will expect
```

```

        //instruction)
        SpiReadWrite(RRC | 0x09);           // Read R_RF_CONFIG bytes
                                           //starting at 09 (UP_CLK_FREQ)
        cklf = SpiReadWrite(0) | 0x04;     // Set XOF to 001 (0x04 - 16MHz)
        RACSN = 1;                         // Set CSN on the radio back to low before next
                                           //instruction (another high to low transition is
                                           //needed thus the next line)
        RACSN = 0;                         // Back to low, radio expects another intruction
        SpiReadWrite(WRC | 0x09);         // Instruct SPI to write RF_CONFIG
        SpiReadWrite(cklf);               // Write RF_CONFIG
        RACSN = 1;                         // Reset CSN to high
    }

// Initialize timer used for UART clocking
void InitUartTimer1(void)
{
    TH1 = 243;                            // 19200@16MHz (when T1M=1 and SMOD=1)
    CKCON |= 0x10;                        // T1M=1 (/4 timer clock)
    PCON = 0x80;                          // SMOD=1 (double baud rate)
    SCON = 0x52;                          // Serial model, enable receiver
    TMOD = 0x20;                          // Timer1 8bit auto reload
    TR1 = 1;                              // Start timer1
    P0_ALT |= 0x06;                       // Select alternate functions on pins
                                           //P0.1 and P0.2
    P0_DIR |= 0x02;                       // P0.1 (RxD) is input

    SPICLK = 0;                           // Max SPI clock
    SPI_CTRL = 0x02;                       // Connect internal SPI controller to
                                           //Radio
    ES = 0;
}

// Sleep function
void longDelay()
{
    unsigned int i;
    unsigned int n = 0xFFFF;
    while(n--)
        for(i=0; i<0xFFFF; i++)
            ;
}

// Changes Receiving address of a node
void ChangeRXAddress(unsigned int xdata *RXAddr)
{
    unsigned int i;

    RACSN = 0;
    SpiReadWrite(WRC | 0x05);             //Write to RFConfig starting at byte 5
                                           //(RF Address)
    for(i=0; i<4; i++)
        SpiReadWrite(RXAddr[i]);
    RACSN = 1;
}

// Initialises radio transceiver on channel 0x68
void InitRadio(void)
{
    TXEN = 0;
    TRX_CE = 0;
    RACSN = 0;
}

```

```

    SpiReadWrite(CC | (POWER << 2) | (HFREQ << 1) | (0x00)); //pass
    //first 8 bits to the register (including channel high bit)
    SpiReadWrite(0x68); //pass low 8 bits of the channel
    RACSN = 1;
    EA = 1; //Global enable for all interrupts
}

// Transmits a 32-byte packet over the radio
void TransmitBytes(unsigned char data *TXAddr, unsigned char xdata *buff)
{
    unsigned char i;

    longDelay(); //Wait before transmitting

    //Configure TX Address
    RACSN = 0;
    SpiReadWrite(WTA); //Write to RFConfig starting at byte 5 (RF
    //Address)

    for(i=0; i<4; i++)
        SpiReadWrite(TXAddr[i]);
    RACSN = 1;

    //Write 32-byte packet to SPI
    RACSN = 0;
    SpiReadWrite(WTP); // write packet to SPI
    for (i=0; i<32; i++)
    {
        SpiReadWrite(buff[i]);
    }
    RACSN = 1;

    //wait until channel is clear
    while(CD == 1)
        ;

    TRX_CE = 1; // enable radio
    TXEN = 1; // enable radio TX mode
    while(DR == 0) // wait until data ready goes high
        ;

    TRX_CE = 0; // disable radio
    TXEN = 0; // disable TX mode
}

// Receives 32-byte packet if AM (Address Match) flag is raised
unsigned char ReceiveMode(unsigned char xdata *buff)
{
    unsigned char i;
    unsigned char j;
    unsigned char amFlag;

    amFlag = 0; //reset address match flag
    TXEN = 0; //Set TX_EN to low to enter Shockburst receive mode

    TRX_CE = 1; //enable radio

    j=0; //wait for Carrier Detect
    while(CD == 0 && j<255)
        j++;

    if (AM) //If Address Match: process the SPI buffer

```

```

{
    while(DR == 0)          //Wait until Data Ready
        ;

    RACSN = 0;
    SpiReadWrite(RRP);     //Send packet read command to the SPI
    for (i = 0; i < 32; i++) //Read in the packet of the buffer
        buff[i] = SpiReadWrite(0);
    RACSN = 1;

    TRX_CE=0;             //disable RX mode
    PutString("\r\nRX\r\n");

    amFlag = 1;
    return amFlag;
}
return amFlag;
}

// Integer to ASCII (itoa) conversion
void itoa(int n, unsigned char *s)
{
    unsigned char *charPtr;
    int idata n1;
    unsigned char idata len;

    len=0;
    //change the sign for negative numbers
    if (n<0)
    {
        n=-n;
        *s++ = '-';
    }
    //calculate the length of the number in decimal digits
    n1=n;
    do
    {
        n1 /= 10;
        len++;
    }
    while(n1);

    *(charPtr = &s[len]) = 0; //null terminate string
    do
    {
        *--charPtr = (n % 10) + '0';
        n /= 10;
    }
    while(n);
}

// Prints array of 12 bytes in decimal notation
void printArray (unsigned char idata *array)
{
    unsigned char i;
    unsigned char stringBuffer[5];

    for (i=0; i<12; i++)
    {

```



```

        itoa((int)array[i], stringBuffer);
        PutString(stringBuff);
        PutString("\t");
    }
    PutString("\r\n");
}

//Returns Modulo96 of the 12-byte number (Big Endian) held in 12-element
//array
unsigned char getModulo96 (unsigned char idata *array)
{
    unsigned char i;
    unsigned char modulus = 0;

    for (i=0; i<11; i++)
    {
        //Divide and conquer approach: sum of two 4-bit numbers multiplied
        //by 2*16
        modulus += (( array[i] & 0x0f) + (array[i] >> 4) ) * 256) % 96;
        modulus %= 96; //Reduce each result Mod96 - can be done less
        //frequently
    }
    return (array[11] + modulus) % 96; //Add the result to the LSB and
    //calculate Mod96 again
}

//Reverses (mirror) the array - used by indexShift function
void arrayReverse(unsigned char idata *result, unsigned char left, unsigned
char right)
{
    unsigned char temp;
    unsigned char i;
    unsigned char j;

    //Start with edges and continue until middle elements are processed
    for (i=left, j=right; i<j; i++, j--)
    {
        temp = result[i];
        result[i] = result[j];
        result[j] = temp;
    }
}

//Rotates the elements of a 12-element array by up to 11 positions left or
right
void indexShift (unsigned char idata *result, unsigned int direction,
unsigned int indexShift)
{
    //ArrayReverse: let array be split into A.B. After rotations it is B.A
    //B.A = reverse( reverse(A).reverse(B) )

    //Direction: 0 for left shift, 1 for right shift
    if (!direction)
    {
        arrayReverse(result, 0, indexShift-1);
        arrayReverse(result, indexShift, 11);
        arrayReverse(result, 0, 11);
    }
    else
    {
        arrayReverse(result, 12-indexShift, 11);
    }
}

```

```

        arrayReverse(result, 0, 11-indexShift);
        arrayReverse(result, 0, 11);
    }
}

//Bitwise bit rotation of the array (up to 7 places)
void bitShift (unsigned char idata *array, unsigned int direction, unsigned
int bitsToShift)
{
    unsigned char i;
    unsigned char element0;
    unsigned char temp;

    //Direction: 0 for left shift, 1 for right shift
    if (!direction)//Shift bits to the left with carry to the lower element
    {
        element0 = array[0];
        array[0] = array[0] << bitsToShift;

        for (i=0; i<11; i++)
        {
            temp = array[i+1];
            array[i+1] = array[i+1] << bitsToShift;
            array[i] |= temp >> (8 - bitsToShift);
        }
        array[11] |= element0 >> (8-bitsToShift);
    }
    else //Shift bits to the right with carry to the lower element
    {
        element0 = array[11];
        array[11] = array[11] >> bitsToShift;

        for (i=11; i>0; i--)
        {
            temp = array[i-1];
            array[i-1] = array[i-1] >> bitsToShift;
            array[i] |= temp << (8 - bitsToShift);
        }
        array[0] |= element0 << (8-bitsToShift);
    }
}

//Rotates array2 by array1 Modulo96
void bitRotation (unsigned char idata *array1, unsigned char idata *result,
unsigned int direction)
{
    unsigned char modulo = getModulo96(array1); //First get modulo
    unsigned char indicesToShift;
    unsigned char bitsToShift;

    //Second divide modulo by 8 and rotate the array (if modulo is bigger
    //than 8)
    //Direction: 0 for left shift, 1 for right shift
    if (modulo > 8)
    {
        indicesToShift = modulo/8;
        indexShift(result, direction, indicesToShift);
    }

    //Then bitshift with carry each element by the remaining shift (shift

```

```

    //amount will be <8)
    bitsToShift = modulo%8;
    if (bitsToShift != 0)
        bitShift(result, direction, bitsToShift);
}

//Performs XOR on two arrays and saves the output to the second argument
void xorArrays (unsigned char idata *array1, unsigned char idata *result)
{
    unsigned char i;

    for (i=0; i<12; i++)
    {
        result[i] ^= array1[i];
    }
}

//Performs addition Modulo96 on two arrays and saves the output to the
second argument
void additionMod96 (unsigned char idata *array1, unsigned char idata
*result)
{
    unsigned char i;
    unsigned char j;

    for (i=11; i>0; i--)
    {
        result[i] += array1[i]; //Add two bytes (no carry)
        if (result[i] < array1[i]) //Check if carry needed and append to
            //upper byte
        {
            result[i-1]++;
            //check if previous byte was not 255 overloaded to 0 and step
            //back to lower elements to do the same
            j=i;
            //If a carry bit overloads upper byte increment upper to
            //the overloaded one
            //Continue until the array head is met if needed
            while(result[j-1] == 0 && j > 1)
            {
                result[j-2]++;
                --j;
            }
        }
        result[0] += array1[0]; //Got to the MSB - just add and ignore carry
    }
}

//Performs Gossamer MixBits function on two arrays and returns pointer to a
//temporary array
unsigned char* mixBits (unsigned char idata *array1, unsigned char idata
*array2)
{
    // Z = mixBits (X,Y)
    // Z = X
    // 32times: Z = (Z>>1) + Z + Z + Y
    unsigned char idata result[12];
    unsigned char i;

    for (i=0; i<12; i++)

```

```

    {
        result[i] = array1[i];
    }

    for (i=0; i<32; i++)
    {
        bitShift (array1, 1, 1);
        additionMod96 (array1, result);
        bitShift (array1, 0, 1);
        additionMod96 (array1, result);
        additionMod96 (array1, result);
        additionMod96 (array2, result);
    }
    return result;
}

/* SEA S-Box implementation according to SEA author's suggestions
void seaSBOX (unsigned char data *block, unsigned char i)
{
    block[3*i] = (block[3*i+2] && block[3*i+1]) ^ block[3*i];
    block[3*i+1] = (block[3*i+2] && block[3*i]) ^ block[3*i+1];
    block[3*i+2] = (block[3*i] || block[3*i+1]) ^ block[3*i+2];
}
*/

//Simplified S-Box - per private conversation with the author it is safe to
//perform S-Box on the first three elements only (SEA(96,8).
//Originally author advised to apply S-Box to any 3 elements of each block
void seaSBOX (unsigned char idata *block, unsigned char i)
{
    block[0] = (block[2] && block[1]) ^ block[0];
    block[1] = (block[2] && block[0]) ^ block[1];
    block[2] = (block[0] || block[1]) ^ block[2];
}

//SEA Bit-rotation function for SEA(96,8).
//Function uses Raisonance RC51 intrinsic functions (_cror_ and _crol_).
void seaBitRotation (unsigned char idata *block)
{
    block[0] = _cror_(block[0], 1);
    block[2] = _crol_(block[2], 1);
    block[3] = _cror_(block[3], 1);
    block[5] = _crol_(block[5], 1);
}

//SEA(96,8) word rotation - rotates the array by one byte
void seaWordRotation (unsigned char idata *block, unsigned char direction)
{
    //Direction 0 for left and 1 for right rotation
    unsigned char i;
    unsigned char temp;

    if (direction == 0)
    {
        temp = block[0];
        for (i=0; i<5; i++)
            block[i] = block[i+1];
        block[5] = temp;
    }
    else
    {

```

```

    temp = block[5];
    for (i=5; i>0; i--)
        block[i] = block[i-1];
    block[0] = temp;
}
}

//Performs one SEA(96,8) Crypto round. Parameter direction: 0 for
encryption and 1 for decryption
void seaCryptRound (unsigned char direction, unsigned char idata
*blockLeft, unsigned char idata *blockRight, unsigned char idata *keyHalf)
{
    unsigned char i;
    unsigned char temp[6];

    //Every operation will be performed on blockLeft as this memory
    //location will become
    //a right block for the next round.
    //Save the left block
    for (i=0; i<6; i++)
        temp[i] = blockRight[i];

    //ENCRYPTION
    //Fe(Li, Ri, K/2i) <=> RightWordRot(Li) XOR bitRot(sbox(Ri+K/2i))
    //DECRYPTION
    //Fd(Li, Ri, K/2i) <=> LeftWordRot(Li XOR bitRot(sbox(Ri+K/2i)))

    //Step by step:
    //Ri+K/2i
    for (i=0; i<6; i++)
        blockRight[i] += keyHalf[i];
    //sbox(Ri+K/2i)
    seaSBOX(blockRight, i%2);
    ///seaSBOX(blockRight, 0);
    //bitRot(sbox(Ri+K/2i))
    seaBitRotation(blockRight);
    //RightWordRot(Li) - encryption only
    //Direction 0 for encryption and 1 for decryption
    if (direction == 0)
        seaWordRotation(blockLeft, 1);
    //RightWordRot(Li) XOR bitRot(sbox(Ri+K/2i))
    for (i=0; i<6; i++)
    {
        blockRight[i] ^= blockLeft[i];
        blockLeft[i] = temp[i]; //BlockLeft(i)+1 becomes BlockRight(i)
    }
    //LeftWordRot(Li XOR bitRot(sbox(Ri+K/2i))) - decryption only
    if (direction == 1)
        seaWordRotation(blockRight, 0);
}

//Performs one SEA(96,8) key round.
void seaKeyRound (unsigned char idata *keyLeft, unsigned char idata
*keyRight, unsigned char Ci)
{
    //Fk(KLi-1,KRi-1,Ci) <=> KRi = KLi-1 XOR RightWordRot(bitRot(sbox((KRi-
//1)+Ci)));
    unsigned char i;
    unsigned char temp[6];

    //Save the left key (left key will become right after the round)

```

```

//Every operation will be performed on keyLeft as this memory location
//will become a right key
//for the next round.
for (i=0; i<6; i++)
    temp[i] = keyRight[i];
//Step-by-step:
//init Ci (LSW equals i)
///Ci[5] = i;
//(KRi-1)+Ci
keyRight[5] += Ci;
//sbox((KRi-1)+Ci)
seaSBOX(keyRight, (Ci%2));
////seaSBOX(keyRight, 1);
//bitRotation(sbox((KRi-1)+Ci))
seaBitRotation(keyRight);
//RightWordRot(bitRot(sbox((KRi-1)+Ci)));
seaWordRotation(keyRight, 1);
//KRi = KLi-1 XOR RightWordRot(bitRot(sbox((KRi-1)+Ci)));
for (i=0; i<6; i++)
{
    keyRight[i] ^= keyLeft[i];
    keyLeft[i] = temp[i];    //KeyLeft(i)+1 becomes KeyRight(i)
}
}

// SEA Scalable Encryption Algorithm (SEA 96,8) implementation
void sea (unsigned char direction, unsigned char idata *block, unsigned
char idata *key)
{
    //Direction 0 for encryption and 1 for decryption
    unsigned char i;

    //initialization
    unsigned char* idata keyLeft = &key[0];
    unsigned char* idata keyRight = &key[6];
    unsigned char* idata blockLeft = &block[0];
    unsigned char* idata blockRight = &block[6];
    unsigned char* idata temp;    //temp pointer used for swapping key sides
    unsigned char tmp;

    //First half of all rounds (93 as per author's recommendation for a
    //minimum number of rounds)
    //for (i=1; i<47; i++)
    for (i=1; i<47; i++)
    {
        //Key scheduling
        //[KLi, KRi] = Fk(KLi-1, KRi-1, C(i));
        //Fk(KLi-1, KRi-1, Ci) <=> KRi = KLi-1 XOR Rot(bitRot(sbox((KRi-
        //1)+Ci)));
        seaCryptRound (direction, (unsigned char idata *)blockLeft,
(unsigned char idata *)blockRight, (unsigned char idata *)keyRight);
        seaKeyRound((unsigned char idata *)keyLeft, (unsigned char idata
*)keyRight, i);
    }

    //End of round half - swap pointers
    temp = keyLeft;
    keyLeft = keyRight;
    keyRight = temp;

    //for (i=46; i<92; i++)

```

```

for (i=46; i>0; i--)
{
    //Key scheduling part 2
    //[KLi, KRi] = Fk(KLi-1, KRi-1, C(r-i));
    //Fk(KLi-1, KRi-1, Ci) <=> KRi = KLi-1 XOR Rot(bitRot(sbox((KRi-
        //1)+Ci)))
    seaCryptRound (direction, (unsigned char idata *)blockLeft,
(unsigned char idata *)blockRight, (unsigned char idata *)keyLeft);
    seaKeyRound((unsigned char idata *)keyLeft, (unsigned char idata
*)keyRight, i);
}
    seaCryptRound (direction, (unsigned char idata *)blockLeft, (unsigned
char idata *)blockRight, (unsigned char idata *)keyLeft);

//Final: switch Key and Block halves
//indexShift (block, 0, 6); //Gossamer function may be used to save
//space
for(i=0; i<6; i++)
{
    tmp = block[i];
    block[i] = block[i+6];
    block[i+6] = tmp;
}
}

//Copies array to the location of the second argument
void copyArray(unsigned char idata *source, unsigned char idata *target)
{
    unsigned char i;

    for (i=0; i<12; i++)
        target[i] = source[i];
}

//Main Gossamer Master loop - simplified model.
void gossamerMaster (void)
{
    //Simplifications:
    //All data stored in RAM (idata)
    //n1 and n2 random numbers are hardcoded;
    //IDS, k1 and k2 for an example slave device are also hardcoded;
    unsigned char data TXaddr[4] = { 0xC4, 0x5A, 0x5A, 0xC4 };
    unsigned char xdata TXbuff[32];
    unsigned char xdata RXbuff[32];
    unsigned char i;
    unsigned char flag;

    unsigned char idata temp[12];
    unsigned char idata nltemp[12];
    unsigned char *tempPtr;

    unsigned char idata Pi[12] = { 0x32, 0x43, 0xF6, 0xA8, 0x88, 0x5A,
0x30, 0x8D, 0x31, 0x31, 0x98, 0xA2 };
    unsigned char idata IDS[12] = { 0x01, 0x01, 0x01, 0x01, 0x01, 0x01,
0x01, 0x01, 0x01, 0x01, 0x01, 0x01 };
    unsigned char idata ID[12] = { 0x44, 0x44, 0x44, 0x44, 0x44, 0x44,
0x44, 0x44, 0x44, 0x44, 0x44, 0x44 };
    unsigned char idata k1[12] = { 0x10, 0x10, 0x10, 0x10, 0x10, 0x10,
0x10, 0x10, 0x10, 0x10, 0x10, 0x10 };
    unsigned char idata k2[12] = { 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
0x20, 0x20, 0x20, 0x20, 0x20, 0x20 };
}

```

```

    unsigned char idata n1[12] = { 0x22, 0x22, 0x22, 0x22, 0x22, 0x22,
    0x22, 0x22, 0x22, 0x22, 0x22, 0x22 };
    unsigned char idata n2[12] = { 0x23, 0x23, 0x23, 0x23, 0x23, 0x23,
    0x23, 0x23, 0x23, 0x23, 0x23, 0x23 };

    unsigned char idata messageA[12];
    unsigned char idata messageB[12];
    unsigned char idata messageC[12];
    unsigned char idata messageD[12];
    unsigned char idata k1next[12];
    unsigned char idata k2next[12];

    //Loop forever authenticating the experimental slave
    for (;;)
    {
        //Tag identification: verify incoming IDS
        flag = 1;
        while (!ReceiveMode(RXbuff))
            ;
        for (i=0; i<12; i++)
        {
            if (IDS[i] != RXbuff[i])
                flag = 0;
        }
        if (flag) //If IDS is correct continue with the protocol
        {
            PutString("\r\nIDS OK");
            //Create message A: A = ROT((ROT(IDS+k1+Pi+n1, k2)+k1, k1)
            //messageA + n1
            copyArray(n1, messageA);
            //n1+Pi
            additionMod96(Pi, messageA);
            //k1+Pi+n1
            additionMod96(k1, messageA);
            //IDS+k1+Pi+n1
            additionMod96(IDS, messageA);
            //ROT(IDS+k1+Pi+n1, k2)
            bitRotation(k2, messageA, 0);
            //ROT(IDS+k1+Pi+n1, k2)+k1
            additionMod96(k1, messageA);
            //ROT((ROT(IDS+k1+Pi+n1, k2)+k1, k1)
            bitRotation(k1, messageA, 0);

            //MessageA created - now transmit:
            PutString("\r\nA\t");
            printArray(messageA);
            for (i=0; i<12; i++)
                TXbuff[i] = messageA[i];
            TransmitBytes(TXaddr, TXbuff);

            //Create message B: B = ROT((ROT(IDS+k2+Pi+n2, k1)+k2, k2)
            //messageB + n2
            copyArray(n2, messageB);
            //Pi+n2
            additionMod96(Pi, messageB);
            //k2+Pi+n2
            additionMod96(k2, messageB);
            //IDS+k2+Pi+n2
            additionMod96(IDS, messageB);
            //ROT(IDS+k2+Pi+n2, k1)
            bitRotation(k1, messageB, 0);
        }
    }

```



```

//ROT(IDS+k2+Pi+n2, k1)+k2
additionMod96(k2, messageB);
//ROT((ROT(IDS+k2+Pi+n2, k1)+k2, k2)
bitRotation (k2, messageB, 0);

//MessageB created - now transmit:
PutString("B\t");
printArray(messageB);
for (i=0; i<12; i++)
    TXbuff[i] = messageB[i];
TransmitBytes(TXaddr, TXbuff);

//Create temporary n3: n3 = mixBits(n1,n2)
tempPtr = mixBits(n1, n2);
for (i=0; i<12; i++)
    temp[i] = tempPtr[i];

//Create keys for the next round
//k1next = ROT((ROT(n2+k1+Pi+n3, n2)+k2 XOR n3, n1) XOR n3
//k1next + n3
copyArray(temp, k1next);
//Pi+n3
additionMod96(Pi, k1next);
//k1+Pi+n3
additionMod96(k1, k1next);
//n2+k1+Pi+n3
additionMod96(n2, k1next);
//ROT(n2+k1+Pi+n3, n2)
bitRotation (n2, k1next, 0);
//ROT(n2+k1+Pi+n3, n2)+k2
additionMod96(k2, k1next);
//ROT(n2+k1+Pi+n3, n2)+k2 XOR n3
xorArrays(temp, k1next);
//ROT((ROT(n2+k1+Pi+n3, n2)+k2 XOR n3, n1)
bitRotation (n1, k1next, 0);
//ROT((ROT(n2+k1+Pi+n3, n2)+k2 XOR n3, n1) XOR n3
xorArrays(temp, k1next);

//k2next = ROT((ROT(n1+k2+Pi+n3, n1)+k1+n3, n2)+n3
//k2next+n3
copyArray(temp, k2next);
//Pi+n3
additionMod96(Pi, k2next);
//k2+Pi+n3
additionMod96(k2, k2next);
//n1+k2+Pi+n3
additionMod96(n1, k2next);
//ROT(n1+k2+Pi+n3, n1)
bitRotation (n1, k2next, 0);
//ROT(n1+k2+Pi+n3, n1)+k1
additionMod96(k1, k2next);
//ROT(n1+k2+Pi+n3, n1)+k1+n3
additionMod96(temp, k2next);
//ROT((ROT(n1+k2+Pi+n3, n1)+k1+n3, n2)
bitRotation (n2, k2next, 0);
//ROT((ROT(n1+k2+Pi+n3, n1)+k1+n3, n2)+n3
additionMod96(temp, k2next);

//Create temporary n1' = mixBits(n3, n2)
tempPtr = mixBits(temp, n2);
for (i=0; i<12; i++)

```

```

    nltemp[i] = tempPtr[i];

//Create message C: C = ROT((ROT(n3+k1next+Pi+n1', n3)+k2next
//XOR n1', n2) XOR n1'
//messageC+n1'
copyArray(nltemp, messageC);
//Pi+n1'
additionMod96(Pi, messageC);
//k1next+Pi+n1'
additionMod96(k1next, messageC);
//n3+k1next+Pi+n1'
additionMod96(temp, messageC);
//ROT(n3+k1next+Pi+n1', n3)
bitRotation (temp, messageC, 0);
//ROT(n3+k1next+Pi+n1', n3)+k2next
additionMod96(k2next, messageC);
//ROT(n3+k1next+Pi+n1', n3)+k2next XOR n1'
xorArrays(nltemp, messageC);
//ROT((ROT(n3+k1next+Pi+n1', n3)+k2next XOR n1', n2)
bitRotation (n2, messageC, 0);
//ROT((ROT(n3+k1next+Pi+n1', n3)+k2next XOR n1', n2) XOR n1'
xorArrays(nltemp, messageC);

//MessageC created - now transmit:
PutString("C\t");
printArray(messageC);
for (i=0; i<12; i++)
    TXbuff[i] = messageC[i];
TransmitBytes(TXaddr, TXbuff);

//Now awaiting reply (message D) - entering receive mode
while (!ReceiveMode(RXbuff))
    ;
//Got message D - verify if successfull
//Step1: calculate local messageD
//D = ROT(ROT(n2+k2next+ID+n1', n2)+k1next+n1', n3)+n1'
//messageD +n1'
copyArray(nltemp, messageD);
//ID+n1'
additionMod96(ID, messageD);
//k2next+ID+n1'
additionMod96(k2next, messageD);
//n2+k2next+ID+n1'
additionMod96(n2, messageD);
//ROT(n2+k2next+ID+n1', n2)
bitRotation (n2, messageD, 0);
//ROT(n2+k2next+ID+n1', n2)+k1next
additionMod96(k1next, messageD);
//ROT(n2+k2next+ID+n1', n2)+k1next+n1'
additionMod96(nltemp, messageD);
//ROT(ROT(n2+k2next+ID+n1', n2)+k1next+n1', n3)
bitRotation (temp, messageD, 0);
//ROT(ROT(n2+k2next+ID+n1', n2)+k1next+n1', n3)+n1'
additionMod96(nltemp, messageD);

//Now verify message D received with a local copy
for (i=0; i<12; i++)
{
    if (messageD[i] != RXbuff[i])
        flag = 0;
}

```

```

if(flag) //Message D matches - key and IDS updating phase
{
    //n2 array will be reused
    //n2' = mixBits(n1', n3)
    tempPtr = mixBits(n1temp, temp);
    for (i=0; i<12; i++)
        n2[i] = tempPtr[i];
    //IDS = ROT((ROT(n1'+k1next+IDS+n2', n1')+k2next XOR n2',
    //n3) XOR n2
    //IDS+n2'
    copyArray(n2, IDS);
    //k1next+IDS+n2'
    additionMod96(k1next, IDS);
    //n1'+k1next+IDS+n2'
    additionMod96(n1temp, IDS);
    //ROT(n1'+k1next+IDS+n2', n1')
    bitRotation (n1temp, IDS, 0);
    //ROT(n1'+k1next+IDS+n2', n1')+k2next
    additionMod96(k2next, IDS);
    //ROT(n1'+k1next+IDS+n2', n1')+k2next XOR n2'
    xorArrays (n2, IDS);
    //ROT((ROT(n1'+k1next+IDS+n2', n1')+k2next XOR n2', n3)
    bitRotation (temp, IDS, 0);
    //ROT((ROT(n1'+k1next+IDS+n2', n1')+k2next XOR n2', n3) XOR
    //n2
    xorArrays (n2, IDS);
    PutString("nIDS\t");
    printArray (IDS);

    //k1 update
    //k1 = ROT((ROT(n3+k2next+Pi+n2', n3)+k1next+n2', n1')+n2'
    //k1=n2'
    copyArray(n2, k1);
    //Pi+n2'
    additionMod96(Pi, k1);
    //k2next+Pi+n2'
    additionMod96(k2next, k1);
    //n3+k2next+Pi+n2'
    additionMod96(temp, k1);
    //ROT(n3+k2next+Pi+n2', n3)
    bitRotation (temp, k1, 0);
    //ROT(n3+k2next+Pi+n2', n3)+k1next
    additionMod96(k1next, k1);
    //ROT(n3+k2next+Pi+n2', n3)+k1next+n2'
    additionMod96(n2, k1);
    //ROT((ROT(n3+k2next+Pi+n2', n3)+k1next+n2', n1')
    bitRotation (n1temp, k1, 0);
    //ROT((ROT(n3+k2next+Pi+n2', n3)+k1next+n2', n1')+n2'
    additionMod96(n2, k1);

    //k2 update
    //k2 = ROT((ROT(IDS+k2next+Pi+k1, IDS)+k1next+k1, n2')+k1
    //k2 = k1
    copyArray(k1, k2);
    //k1+Pi
    additionMod96(Pi, k2);
    //k2next+Pi+k1
    additionMod96(k2next, k2);
    //IDS+k2next+Pi+k1
    additionMod96(IDS, k2);
}

```

```

//ROT(IDS+k2next+Pi+k1, IDS)
bitRotation (IDS, k2, 0);
//ROT(IDS+k2next+Pi+k1, IDS)+k1next
additionMod96(k1next, k2);
//ROT(IDS+k2next+Pi+k1, IDS)+k1next+k1
additionMod96(k1, k2);
//ROT((ROT(IDS+k2next+Pi+k1, IDS)+k1next+k1, n2'))
bitRotation (n2, k2, 0);
//ROT((ROT(IDS+k2next+Pi+k1, IDS)+k1next+k1, n2'))+k1
additionMod96(k1, k2);

//SEA Demonstration: encrypt temp with k1 and send to the
//slave
PutString("SeaD\t");
printArray(temp);
sea(0, temp, k1);
PutString("SeaE\t");
printArray(temp);
PutString("k1\t");
printArray(k1);

for (i=0; i<12; i++)
    TXbuff[i] = temp[i];
TransmitBytes(TXaddr, TXbuff);
}
else
{
    PutString(":( D"); //Incorrect message D received
}
}
else
{
    PutString(":( IDS"); //Incorrect IDS received
}
}
}

//Main function - intitialisation of the NRF9E5
int main(void)
{
    unsigned int xdata RXaddr[4] = { 0xC3, 0x5A, 0x5A, 0xC3 };

    InitUartTimer1();
    SetClock();
    ChangeRXAddress(RXaddr);
    InitRadio();
    PutString(":\n\r");
    gossamerMaster();

    return 0;
}

```

```

/*****
Copyright 2010 Piotr Ksiazak
Filename: Slave.c
Project : MSc - IWSN Experimental Slave
*****/
Version 1.0: Initial release
*****/

#include <reg9e5.h>
#include <intri51.h>
#define POWER      3           // 0=min power...3 = max power
#define HFREQ      1           // 0=433MHz, 1=868/915MHz
#define CHANNEL    351        // Channel number: f(MHz) =
                               // (422.4+CHANNEL/10)*(1+HFREQ)
#pragma REGPARMS              //pass arguments to registers

// SPI access
unsigned char SpiReadWrite(unsigned char b)
{
    EXIF &= ~0x20;           // Clear SPI interrupt
    SPI_DATA = b;           // Move byte to send to SPI data
                               //register
    while((EXIF & 0x20) == 0x00) // Wait until SPI hs finished
                               //transmitting
        ;
    return SPI_DATA;
}

// Send character to UART
void PutChar(char c)
{
    while(!TI)
        ;
    TI = 0;
    SBUF = c;
}

// Read character from UART
unsigned char GetChar(void)
{
    while(!RI)
        ;
    RI = 0;
    return SBUF;
}

// Send string to UART
void PutString(const char *s)
{
    while(*s != 0)
        PutChar(*s++);
}

// Switch to 16MHz clock:
void SetClock(void)
{
    unsigned char cklf;

    RACSN = 0;           // Set CSN on the radio to low (Radio will
                          //expect instruction)
}

```

```

    SpiReadWrite(RRC | 0x09); // Read R_RF_CONFIG bytes starting at 09
                               // (UP_CLK_FREQ)
    cklf = SpiReadWrite(0) | 0x04; // Set XOF to 001 (0x04 - 16MHz)
    RACSN = 1; // Set CSN on the radio back to low before next
//instruction (another high to low transition is needed thus the next line)
    RACSN = 0; // Back to low, radio expects another instruction
    SpiReadWrite(WRC | 0x09); // Instruct SPI to write RF_CONFIG
    SpiReadWrite(cklf); // Write RF_CONFIG
    RACSN = 1; // Reset CSN to high
}

// Initialize timer used for UART clocking
void InitUartTimer1(void)
{
    TH1 = 243; // 19200@16MHz (when T1M=1 and SMOD=1)
    CKCON |= 0x10; // T1M=1 (/4 timer clock)
    PCON = 0x80; // SMOD=1 (double baud rate)
    SCON = 0x52; // Serial model, enable receiver
    TMOD = 0x20; // Timer1 8bit auto reload
    TR1 = 1; // Start timer1
    P0_ALT |= 0x06; // Select alternate functions on pins
                    // P0.1 and P0.2
    P0_DIR |= 0x02; // P0.1 (RxD) is input

    SPICLK = 0; // Max SPI clock
    SPI_CTRL = 0x02; // Connect internal SPI controller to
                    // Radio
    ES = 0;
}

// Changes Receiving address of a node
void ChangeRXAddress(unsigned int xdata *RXAddr)
{
    unsigned int i;

    RACSN = 0;
    SpiReadWrite(WRC | 0x05); // Write to RFConfig starting at byte 5
                               // (RF Address)
    for(i=0; i<4; i++)
        SpiReadWrite(RXAddr[i]);
    RACSN = 1;
}

// Initialises radio
void InitRadio(void)
{
    TXEN = 0;
    TRX_CE = 0;
    //ChangeChannel(); // (0x00, 0x68);
    RACSN = 0;
    SpiReadWrite(CC | (POWER << 2) | (HFREQ << 1) | (0x00)); // pass first 8
                                                               // bits to the register (including channel high bit)
    SpiReadWrite(0x68); // pass low 8 bits of the channel

    RACSN = 1;
    //Channel changed

    EA = 1; // Global enable for all interrupts
}

// Transmits a 32-byte packet over the radio

```

```

void TransmitBytes(unsigned char data *TXAddr, unsigned char xdata *buff)
{
    unsigned char i;

    //Configure TX Address
    RACSN = 0;
    SpiReadWrite(WTA); //Write to RFConfig starting at
                        //byte 5 (RF Address)

    for(i=0; i<4; i++)
        SpiReadWrite(TXAddr[i]);
    RACSN = 1;

    //Write 32-byte packet to SPI
    RACSN = 0;
    SpiReadWrite(WTP); // write packet to SPI
    for (i=0; i<32; i++)
    {
        SpiReadWrite(buff[i]);
    }
    RACSN = 1;

    //wait until channel is clear
    while(CD == 1)
        ;

    TRX_CE = 1; // enable radio
    TXEN = 1; // enable radio TX mode
    while(DR == 0) // wait until data ready goes high
        ;

    TRX_CE = 0; // disable radio
    TXEN = 0; // disable TX mode
}

// Receives 32-byte packet if AM (Address Match) flag is raised
unsigned char ReceiveMode(unsigned char xdata *buff)
{
    unsigned char i;
    unsigned char j;
    unsigned char amFlag;

    amFlag = 0; //reset address match flag
    TXEN = 0; //Set TX_EN to low to enter Shockburst receive mode

    TRX_CE = 1; //enable radio

    j=0; //wait fo Carrier Detect
    while(CD == 0 && j<255)
        j++;

    if (AM) //If Address Match: process the SPI buffer
    {
        while(DR == 0) //Wait until Data Ready
            ;

        RACSN = 0;
        SpiReadWrite(RRP); //Send packet read command to the SPI
        for (i = 0; i < 32; i++) //Read in the packet ot the buffer
            buff[i] = SpiReadWrite(0);
        RACSN = 1;
    }
}

```

```

        TRX_CE=0;                //disable RX mode
        PutString("\r\nRX\r\n");

        amFlag = 1;
        return amFlag;
    }
    return amFlag;
}

// Integer to ASCII (itoa) conversion
void itoa(int n, unsigned char *s)
{
    unsigned char *charPtr;
    int idata n1;
    unsigned char idata len;

    len=0;
    //change the sign for negative numbers
    if (n<0)
    {
        n=-n;
        *s++ = '-';
    }
    //calculate the length of the number in decimal digits
    n1=n;
    do
    {
        n1 /= 10;
        len++;
    }
    while(n1);

    *(charPtr = &s[len]) = 0; //null terminate string
    do
    {
        *--charPtr = (n % 10) + '0';
        n /= 10;
    }
    while(n);
}

// Prints array of 12 bytes in decimal notation
void printArray (unsigned char idata *array)
{
    unsigned char i;
    unsigned char stringBuffer[5];

    for (i=0; i<12; i++)
    {
        itoa((int)array[i], stringBuffer);
        PutString(stringBuff);
        PutString("\t");
    }
    PutString("\r\n");
}

//Returns Modulo96 of the 12-byte number (Big Endian) held in 12-element
//array

```



```

unsigned char getModulo96 (unsigned char idata *array)
{
    unsigned char i;
    unsigned char modulus = 0;

    for (i=0; i<11; i++)
    {
//Divide and conquer approach: sum of two 4-bit numbers multiplied by 2*16
        modulus += (( array[i] & 0x0f) + (array[i] >> 4) ) * 256 % 96;
        modulus %= 96; //Reduce each result Mod96 - can be done less
                        //frequently
    }
    return (array[11] + modulus) % 96; //Add the result to the LSB and
//calculate Mod96 again
}

//Reverses (mirror) the array - used by indexShift function
void arrayReverse(unsigned char idata *result, unsigned char left, unsigned
char right)
{
    unsigned char temp;
    unsigned char i;
    unsigned char j;

    //Start with edges and continue until middle elements are processed
    for (i=left, j=right; i<j; i++, j--)
    {
        temp = result[i];
        result[i] = result[j];
        result[j] = temp;
    }
}

//Rotates the elements of a 12-element array by up to 11 positions left or
//right
void indexShift (unsigned char idata *result, unsigned int direction,
unsigned int indexShift)
{
    //ArrayReverse: let array be split into A.B. After rotations it is B.A
    //B.A = reverse( reverse(A).reverse(B) )

    //Direction: 0 for left shift, 1 for right shift
    if (!direction)
    {
        arrayReverse(result, 0, indexShift-1);
        arrayReverse(result, indexShift, 11);
        arrayReverse(result, 0, 11);
    }
    else
    {
        arrayReverse(result, 12-indexShift, 11);
        arrayReverse(result, 0, 11-indexShift);
        arrayReverse(result, 0, 11);
    }
}

//Bitwise bit rotation of the array (up to 7 places)
void bitShift (unsigned char idata *array, unsigned int direction, unsigned
int bitsToShift)
{
    unsigned char i;

```

```

unsigned char element0;
unsigned char temp;

//Direction: 0 for left shift, 1 for right shift
if (!direction)//Shift bits to the left with carry to the lower element
{
    element0 = array[0];
    array[0] = array[0] << bitsToShift;

    for (i=0; i<11; i++)
    {
        temp = array[i+1];
        array[i+1] = array[i+1] << bitsToShift;
        array[i] |= temp >> (8 - bitsToShift);
    }
    array[11] |= element0 >> (8-bitsToShift);
}
else //Shift bits to the right with carry to the lower element
{
    element0 = array[11];
    array[11] = array[11] >> bitsToShift;

    for (i=11; i>0; i--)
    {
        temp = array[i-1];
        array[i-1] = array[i-1] >> bitsToShift;
        array[i] |= temp << (8 - bitsToShift);
    }
    array[0] |= element0 << (8-bitsToShift);
}
}

//Rotates array2 by array1 Modulo96
void bitRotation (unsigned char idata *array1, unsigned char idata *result,
unsigned int direction)
{
    unsigned char modulo = getModulo96(array1); //First get modulo
    unsigned char indicesToShift;
    unsigned char bitsToShift;

    //Second divide modulo by 8 and rotate the array (if modulo is bigger
    //than 8)
    //Direction: 0 for left shift, 1 for right shift
    if (modulo > 8)
    {
        indicesToShift = modulo/8;
        indexShift(result, direction, indicesToShift);
    }

    //Then bitshift with carry each element by the remaining shift (shift
    //amount will be <8)
    bitsToShift = modulo%8;
    if (bitsToShift != 0)
        bitShift(result, direction, bitsToShift);
}

//Performs XOR on two arrays and saves the output to the second argument
void xorArrays (unsigned char idata *array1, unsigned char idata *result)
{
    unsigned char i;

```

```

    for (i=0; i<12; i++)
    {
        result[i] ^= array1[i];
    }
}

//Performs addition Modulo96 on two arrays and saves the output to the
//second argument
void additionMod96 (unsigned char idata *array1, unsigned char idata
*result)
{
    unsigned char i;
    unsigned char j;

    for (i=11; i>0; i--)
    {
        result[i] += array1[i]; //Add two bytes (no carry)
        if (result[i] < array1[i]) //Check if carry needed and append to
            //upper byte
        {
            result[i-1]++;
            //check if previous byte was not 255 overloaded to 0 and step
            //back to lower elements to do the same
            j=i;
            //If a carry bit overloads upper byte increment upper to
            //overloaded
            //Continue until the array head is met if needed
            while(result[j-1] == 0 && j > 1)
            {
                result[j-2]++;
                --j;
            }
        }
    }
    result[0] += array1[0]; //Got to the MSB - just add and ignore carry
        //(adding Mod96 anyway)
}

//Performs subtraction Modulo96 on two arrays and saves the output to the
//second argument
void subtractionMod96 (unsigned char idata *array1, unsigned char idata
*result)
{
    unsigned char i;
    unsigned char j;

    for (i=11; i>0; i--)
    {
        if (result[i] < array1[i]) //Verify if the minuend is not smaller
            //than the subtrahend
        {
            result[i-1] -= 0x01; //borrow LSB from the lower element
            j=i;
            //If a borrow bit overloads upper byte decrement upper byte to
            //the overloaded one
            while(result[j-1] == 0xFF && j > 1)
            {
                result[j-2] -= 0x01;
                j--;
            }
        }
    }
}

```

```

    }
    result[i] -= array1[i]; //Subtract (no carry)
}
result[0] -= array1[0]; //Got to the MSB - just add and ignore carry
// (adding Mod96 anyway)
}

//Performs Gossamer MixBits function on two arrays and returns pointer to a
//temporary array
unsigned char* mixBits (unsigned char idata *array1, unsigned char idata
*array2)
{
    // Z = mixBits (X,Y)
    // Z = X
    // 32times: Z = (Z>>1) + Z + Z + Y
    unsigned char idata result[12];
    unsigned char i;

    for (i=0; i<12; i++)
    {
        result[i] = array1[i];
    }

    for (i=0; i<32; i++)
    {
        bitShift (array1, 1, 1);
        additionMod96 (array1, result);
        bitShift (array1, 0, 1);
        additionMod96 (array1, result);
        additionMod96 (array1, result);
        additionMod96 (array2, result);
    }
    return result;
}

/* SEA S-Box implementation according to SEA author's suggestions
void seaSBOX (unsigned char data *block, unsigned char i)
{
    block[3*i] = (block[3*i+2] && block[3*i+1]) ^ block[3*i];
    block[3*i+1] = (block[3*i+2] && block[3*i]) ^ block[3*i+1];
    block[3*i+2] = (block[3*i] || block[3*i+1]) ^ block[3*i+2];
}
*/

//Simplified S-Box - per private conversation with the author it is safe to
//perform S-Box on the first three elements only (SEA(96,8).
//Originally author advised to apply S-Box to any 3 elements of each block
void seaSBOX (unsigned char idata *block, unsigned char i)
{
    block[0] = (block[2] && block[1]) ^ block[0];
    block[1] = (block[2] && block[0]) ^ block[1];
    block[2] = (block[0] || block[1]) ^ block[2];
}

//SEA Bit-rotation function for SEA(96,8).
//Function uses Raisonance RC51 intrinsic functions (_cror_ and _crol_).
void seaBitRotation (unsigned char idata *block)
{
    block[0] = _cror_(block[0], 1);
    block[2] = _crol_(block[2], 1);
    block[3] = _cror_(block[3], 1);
}

```

```

    block[5] = _crol_(block[5], 1);
}

//SEA(96,8) word rotation - rotates the array by one byte
void seaWordRotation (unsigned char idata *block, unsigned char direction)
{
    //Direction 0 for left and 1 for right rotation
    unsigned char i;
    unsigned char temp;

    if (direction == 0)
    {
        temp = block[0];
        for (i=0; i<5; i++)
            block[i] = block[i+1];
        block[5] = temp;
    }
    else
    {
        temp = block[5];
        for (i=5; i>0; i--)
            block[i] = block[i-1];
        block[0] = temp;
    }
}

//Performs one SEA(96,8) Crypto round. Parameter direction: 0 for
//encryption and 1 for decryption
void seaCryptRound (unsigned char direction, unsigned char idata
*blockLeft, unsigned char idata *blockRight, unsigned char idata *keyHalf)
{
    unsigned char i;
    unsigned char temp[6];

    //Every operation will be performed on blockLeft as this memory
    //location will become a right block for the next round.
    //Save the left block
    for (i=0; i<6; i++)
        temp[i] = blockRight[i];

    //ENCRYPTION
    //Fe(Li, Ri, K/2i) <=> RightWordRot(Li) XOR bitRot(sbox(Ri+K/2i))
    //DECRIPTION
    //Fd(Li, Ri, K/2i) <=> LeftWordRot(Li XOR bitRot(sbox(Ri+K/2i)))

    //Step by step:
    //Ri+K/2i
    for (i=0; i<6; i++)
        blockRight[i] += keyHalf[i];
    //sbox(Ri+K/2i)
    seaSBOX(blockRight, i%2);
    ///seaSBOX(blockRight, 0);
    //bitRot(sbox(Ri+K/2i))
    seaBitRotation(blockRight);
    //RightWordRot(Li) - encryption only
    //Direction 0 for encryption and 1 for decryption
    if (direction == 0)
        seaWordRotation(blockLeft, 1);
    //RightWordRot(Li) XOR bitRot(sbox(Ri+K/2i))
    for (i=0; i<6; i++)
    {

```

```

        blockRight[i] ^= blockLeft[i];
        blockLeft[i] = temp[i];    //BlockLeft(i)+1 becomes BlockRight(i)
    }
    //LeftWordRot(Li XOR bitRot(sbox(Ri+K/2i))) - decryption only
    if (direction == 1)
        seaWordRotation(blockRight, 0);
}

//Performs one SEA(96,8) key round.
void seaKeyRound (unsigned char idata *keyLeft, unsigned char idata
*keyRight, unsigned char Ci)
{
    //Fk(KLi-1,KRi-1,Ci) <=> KRi = KLi-1 XOR RightWordRot(bitRot(sbox((KRi-
    //1)+Ci)));
    unsigned char i;
    unsigned char temp[6];

    //Save the left key (left key will become right after the round)
    //Every operation will be performed on keyLeft as this memory location
    //will become a right key for the next round.
    for (i=0; i<6; i++)
        temp[i] = keyRight[i];
    //Step-by-step:
    //init Ci (LSW equals i)
    ///Ci[5] = i;
    //(KRi-1)+Ci
    keyRight[5] += Ci;
    //sbox((KRi-1)+Ci)
    seaSBOX(keyRight, (Ci%2));
    ///seaSBOX(keyRight, 1);
    //bitRotation(sbox((KRi-1)+Ci))
    seaBitRotation(keyRight);
    //RightWordRot(bitRot(sbox((KRi-1)+Ci)));
    seaWordRotation(keyRight, 1);
    //KRi = KLi-1 XOR RightWordRot(bitRot(sbox((KRi-1)+Ci)));
    for (i=0; i<6; i++)
    {
        keyRight[i] ^= keyLeft[i];
        keyLeft[i] = temp[i];    //KeyLeft(i)+1 becomes KeyRight(i)
    }
}

// SEA Scalable Encryption Algorithm (SEA 96,8) implementation
void sea (unsigned char direction, unsigned char idata *block, unsigned
char idata *key)
{
    //Direction 0 for encryption and 1 for decryption
    unsigned char i;

    //initialization
    unsigned char* idata keyLeft = &key[0];
    unsigned char* idata keyRight = &key[6];
    unsigned char* idata blockLeft = &block[0];
    unsigned char* idata blockRight = &block[6];
    unsigned char* idata temp; //temp pointer used for swapping key sides
    unsigned char tmp;

    //First half of all rounds (93 as per author's recommendation for a
    //minimum number of rounds)
    for (i=1; i<47; i++)
    {

```

```

    //Key scheduling
    //[KLi, KRi] = Fk(KLi-1, KRi-1, C(i));
    //Fk(KLi-1, KRi-1, Ci) <=> KRi = KLi-1 XOR Rot(bitRot(sbox((KRi-
        //1)+Ci)));
    seaCryptRound (direction, (unsigned char idata *)blockLeft,
(unsigned char idata *)blockRight, (unsigned char idata *)keyRight);
    seaKeyRound((unsigned char idata *)keyLeft, (unsigned char idata
*)keyRight, i);
    }

//End of round half - swap pointers
temp = keyLeft;
keyLeft = keyRight;
keyRight = temp;

for (i=46; i>0; i--)
{
    //Key scheduling part 2
    //[KLi, KRi] = Fk(KLi-1, KRi-1, C(r-i));
    //Fk(KLi-1, KRi-1, Ci) <=> KRi = KLi-1 XOR Rot(bitRot(sbox((KRi-
        //1)+Ci)))
    seaCryptRound (direction, (unsigned char idata *)blockLeft,
(unsigned char idata *)blockRight, (unsigned char idata *)keyLeft);
    seaKeyRound((unsigned char idata *)keyLeft, (unsigned char idata
*)keyRight, i);
    }
    seaCryptRound (direction, (unsigned char idata *)blockLeft, (unsigned
char idata *)blockRight, (unsigned char idata *)keyLeft);

//Final: switch Key and Block halves
//indexShift (block, 0, 6); //Gossamer function may be used to save
//space
for(i=0; i<6; i++)
{
    tmp = block[i];
    block[i] = block[i+6];
    block[i+6] = tmp;
}
}

//Copies array to the location of the second argument
void copyArray(unsigned char idata *source, unsigned char idata *target)
{
    unsigned char i;

    for (i=0; i<12; i++)
        target[i] = source[i];
}

//Main Gossamer Slave loop - simplified model.
void gossamerSlave (void)
{
    //Simplifications:
    //All data stored in RAM (idata);
    //Roll-back to previous IDS and keys in case of receiving incorrect
    //message C
    //not implemented. Need to copy oldk1 and oldk2 and oldIDS arrays into
    //respective current arrays.
    unsigned char data TXaddr[4] = { 0xC3, 0x5A, 0x5A, 0xC3 };
    unsigned char xdata TXbuff[32];
    unsigned char xdata RXbuff[32];

```

```

unsigned char i;
unsigned char flag;

unsigned char idata temp[12];
unsigned char idata n1temp[12];
unsigned char *tempPtr;

unsigned char idata Pi[12] = { 0x32, 0x43, 0xF6, 0xA8, 0x88, 0x5A,
0x30, 0x8D, 0x31, 0x31, 0x98, 0xA2 };
unsigned char idata IDS[12] = { 0x01, 0x01, 0x01, 0x01, 0x01, 0x01,
0x01, 0x01, 0x01, 0x01, 0x01, 0x01 };
unsigned char idata ID[12] = { 0x44, 0x44, 0x44, 0x44, 0x44, 0x44,
0x44, 0x44, 0x44, 0x44, 0x44, 0x44 };
unsigned char idata k1[12] = { 0x10, 0x10, 0x10, 0x10, 0x10, 0x10,
0x10, 0x10, 0x10, 0x10, 0x10, 0x10 };
unsigned char idata k2[12] = { 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
0x20, 0x20, 0x20, 0x20, 0x20, 0x20 };
unsigned char idata n1[12];
unsigned char idata n2[12];

unsigned char idata oldk1[12];
unsigned char idata oldk2[12];

unsigned char idata messageC[12];
unsigned char idata messageD[12];
unsigned char idata k1next[12];
unsigned char idata k2next[12];

//Loop forever authenticating the experimental Master
for (;;)
{
    //Send IDS
    flag = 1;
    PutString("\r\nIDS\t");
    printArray(IDS);

    for (i=0; i<12; i++)
        TXbuff[i] = IDS[i];
    TransmitBytes(TXaddr, TXbuff);

    //Wait for message A
    while (!ReceiveMode(RXbuff))
        ;
    //message A will be used to extract n1 - save it there
    for (i=0; i<12; i++)
        n1[i] = RXbuff[i];
    PutString("A\t");
    printArray(n1);

    //Wait for message B
    while (!ReceiveMode(RXbuff))
        ;
    //message B will be used to extract n2 - save it there
    for (i=0; i<12; i++)
        n2[i] = RXbuff[i];
    PutString("B\t");
    printArray(n2);

    //Extract n1 from A
    //A = ROT((ROT(IDS+k1+Pi+n1, k2)+k1, k1)
    //rightROT A: rightROT((ROT(IDS+k1+Pi+n1, k2)+k1, k1)

```



```

bitRotation (k1, n1, 1);
//ROT(IDS+k1+Pi+n1, k2)-k1
subtractionMod96(k1, n1);
//rightROT(IDS+k1+Pi+n1, k2)
bitRotation (k2, n1, 1);
//k1+Pi+n1-IDS
subtractionMod96(IDS, n1);
//Pi+n1-k1
subtractionMod96(k1, n1);
//n1-Pi
subtractionMod96(Pi, n1);
//n1 extracted, now process message B

//Extract n2 from B
//B = ROT((ROT(IDS+k2+Pi+n2, k1)+k2, k2)
//rightROT((ROT(IDS+k2+Pi+n2, k1)+k2, k2)
bitRotation(k2, n2, 1);
// (ROT(IDS+k2+Pi+n2, k1)-k2
subtractionMod96(k2, n2);
//right(ROT(IDS+k2+Pi+n2, k1)
bitRotation(k1, n2, 1);
//k2+Pi+n2-IDS
subtractionMod96(IDS, n2);
//Pi+n2-k2
subtractionMod96(k2, n2);
//n2-Pi
subtractionMod96(Pi, n2);
//n2 extracted now calculate message C
PutString("N1\t");
printArray(n1);
PutString("N2\t");
printArray(n2);

//Create temporary n3: n3 = mixBits(n1,n2)
tempPtr = mixBits(n1, n2);
for (i=0; i<12; i++)
    temp[i] = tempPtr[i];

//Create keys for the next round
//k1next = ROT((ROT(n2+k1+Pi+n3, n2)+k2 XOR n3, n1) XOR n3
//k1next + n3
copyArray(temp, k1next);
//Pi+n3
additionMod96(Pi, k1next);
//k1+Pi+n3
additionMod96(k1, k1next);
//n2+k1+Pi+n3
additionMod96(n2, k1next);
//ROT(n2+k1+Pi+n3, n2)
bitRotation (n2, k1next, 0);
//ROT(n2+k1+Pi+n3, n2)+k2
additionMod96(k2, k1next);
//ROT(n2+k1+Pi+n3, n2)+k2 XOR n3
xorArrays(temp, k1next);
//ROT((ROT(n2+k1+Pi+n3, n2)+k2 XOR n3, n1)
bitRotation (n1, k1next, 0);
//ROT((ROT(n2+k1+Pi+n3, n2)+k2 XOR n3, n1) XOR n3
xorArrays(temp, k1next);

//k2next = ROT((ROT(n1+k2+Pi+n3, n1)+k1+n3, n2)+n3
//k2next+n3

```

```

copyArray(temp, k2next);
//Pi+n3
additionMod96(Pi, k2next);
//k2+Pi+n3
additionMod96(k2, k2next);
//n1+k2+Pi+n3
additionMod96(n1, k2next);
//ROT(n1+k2+Pi+n3, n1)
bitRotation (n1, k2next, 0);
//ROT(n1+k2+Pi+n3, n1)+k1
additionMod96(k1, k2next);
//ROT(n1+k2+Pi+n3, n1)+k1+n3
additionMod96(temp, k2next);
//ROT((ROT(n1+k2+Pi+n3, n1)+k1+n3, n2)
bitRotation (n2, k2next, 0);
//ROT((ROT(n1+k2+Pi+n3, n1)+k1+n3, n2)+n3
additionMod96(temp, k2next);

//Create temporary n1' = mixBits(n3, n2)
tempPtr = mixBits(temp, n2);
for (i=0; i<12; i++)
    n1temp[i] = tempPtr[i];

//Create message C: C = ROT((ROT(n3+k1next+Pi+n1', n3)+k2next XOR
//n1', n2) XOR n1'
//messageC+n1'
copyArray(n1temp, messageC);
//Pi+n1'
additionMod96(Pi, messageC);
//k1next+Pi+n1'
additionMod96(k1next, messageC);
//n3+k1next+Pi+n1'
additionMod96(temp, messageC);
//ROT(n3+k1next+Pi+n1', n3)
bitRotation (temp, messageC, 0);
//ROT(n3+k1next+Pi+n1', n3)+k2next
additionMod96(k2next, messageC);
//ROT(n3+k1next+Pi+n1', n3)+k2next XOR n1'
xorArrays(n1temp, messageC);
//ROT((ROT(n3+k1next+Pi+n1', n3)+k2next XOR n1', n2)
bitRotation (n2, messageC, 0);
//ROT((ROT(n3+k1next+Pi+n1', n3)+k2next XOR n1', n2) XOR n1'
xorArrays(n1temp, messageC);

//MessageC created -now await message C:
while (!ReceiveMode(RXbuff))
    ;
PutString("LC\t");
printArray(messageC);

//Message C received - verify if matches local copy of C
for (i=0; i<12; i++)
{
    if (messageC[i] != RXbuff[i])
        flag = 0;
}

if (flag) //C matches so Master is authenticated. Send message D
{
    //Step1: calculate local messageD
    //D = ROT(ROT(n2+k2next+ID+n1', n2)+k1next+n1', n3)+n1'

```

```

//messageD +n1'
copyArray(n1temp, messageD);
//ID+n1'
additionMod96(ID, messageD);
//k2next+ID+n1'
additionMod96(k2next, messageD);
//n2+k2next+ID+n1'
additionMod96(n2, messageD);
//ROT(n2+k2next+ID+n1', n2)
bitRotation(n2, messageD, 0);
//ROT(n2+k2next+ID+n1', n2)+k1next
additionMod96(k1next, messageD);
//ROT(n2+k2next+ID+n1', n2)+k1next+n1'
additionMod96(n1temp, messageD);
//ROT(ROT(n2+k2next+ID+n1', n2)+k1next+n1', n3)
bitRotation(temp, messageD, 0);
//ROT(ROT(n2+k2next+ID+n1', n2)+k1next+n1', n3)+n1'
additionMod96(n1temp, messageD);

//Now send message D and go to key updating phase
for (i=0; i<12; i++)
    TXbuff[i] = messageD[i];
TransmitBytes(TXaddr, TXbuff);

//Message D sent - key and IDS updating phase
//First step: save old IDS and keys
//MessageD memory location will be re-used for oldIDS
copyArray(IDS, messageD);
copyArray(k1, oldk1);
copyArray(k2, oldk2);

//n2 array will be reused
//n2' = mixBits(n1', n3)
tempPtr = mixBits(n1temp, temp);
for (i=0; i<12; i++)
    n2[i] = tempPtr[i];
//IDS = ROT((ROT(n1'+k1next+IDS+n2', n1')+k2next XOR n2', n3)
//XOR n2
//IDS+n2'
copyArray(n2, IDS);
//k1next+IDS+n2'
additionMod96(k1next, IDS);
//n1'+k1next+IDS+n2'
additionMod96(n1temp, IDS);
//ROT(n1'+k1next+IDS+n2', n1')
bitRotation(n1temp, IDS, 0);
//ROT(n1'+k1next+IDS+n2', n1')+k2next
additionMod96(k2next, IDS);
//ROT(n1'+k1next+IDS+n2', n1')+k2next XOR n2'
xorArrays(n2, IDS);
//ROT((ROT(n1'+k1next+IDS+n2', n1')+k2next XOR n2', n3)
bitRotation(temp, IDS, 0);
//ROT((ROT(n1'+k1next+IDS+n2', n1')+k2next XOR n2', n3) XOR n2
xorArrays(n2, IDS);

//k1 update
//k1 = ROT((ROT(n3+k2next+Pi+n2', n3)+k1next+n2', n1')+n2'
//k1=n2'
copyArray(n2, k1);
//Pi+n2'
additionMod96(Pi, k1);

```

```

//k2next+Pi+n2'
additionMod96(k2next, k1);
//n3+k2next+Pi+n2'
additionMod96(temp, k1);
//ROT(n3+k2next+Pi+n2', n3)
bitRotation (temp, k1, 0);
//ROT(n3+k2next+Pi+n2', n3)+k1next
additionMod96(k1next, k1);
//ROT(n3+k2next+Pi+n2', n3)+k1next+n2'
additionMod96(n2, k1);
//ROT((ROT(n3+k2next+Pi+n2', n3)+k1next+n2', n1')
bitRotation (n1temp, k1, 0);
//ROT((ROT(n3+k2next+Pi+n2', n3)+k1next+n2', n1')+n2'
additionMod96(n2, k1);

//k2 update
//k2 = ROT((ROT(IDS+k2next+Pi+k1, IDS)+k1next+k1, n2')+k1
//k2 = k1
copyArray(k1, k2);
//k1+Pi
additionMod96(Pi, k2);
//k2next+Pi+k1
additionMod96(k2next, k2);
//IDS+k2next+Pi+k1
additionMod96(IDS, k2);
//ROT(IDS+k2next+Pi+k1, IDS)
bitRotation (IDS, k2, 0);
//ROT(IDS+k2next+Pi+k1, IDS)+k1next
additionMod96(k1next, k2);
//ROT(IDS+k2next+Pi+k1, IDS)+k1next+k1
additionMod96(k1, k2);
//ROT((ROT(IDS+k2next+Pi+k1, IDS)+k1next+k1, n2')
bitRotation (n2, k2, 0);
//ROT((ROT(IDS+k2next+Pi+k1, IDS)+k1next+k1, n2')+k1
additionMod96(k1, k2);

//Get encrypted message from Master and save to temp
while (!ReceiveMode(RXbuff))
    ;
//encrypted message received
for (i=0; i<12; i++)
    temp[i] = RXbuff[i];

//Now decrypt temp with k1 and display
PutString("SeaE\t");
printArray(temp);
sea(1, temp, k1);
PutString("SeaD\t");
printArray(temp);
PutString("k1\t");
printArray(k1);
}
else
    //Simplification: roll-back to old IDS and keys not implemented
    PutString("\r\n:( C"); //wrong message C
}
}

int main(void)
{
    unsigned int xdata RXaddr[4] = { 0xC4, 0x5A, 0x5A, 0xC4 };

```

```
    InitUartTimer1 ();
    SetClock ();
    ChangeRXAddress (RXaddr);
    InitRadio ();
    PutString (":)\n\r");
    gossamerSlave ();

    return 0;
}
```