Reconfigurable Feedback Shift Register Cipher Design and Secure Link Layer Protocol for Wireless Sensor Network

by

Guang Zeng
B.Sc., Beijing University of Posts and Telecommunications, 2011

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

Reconfigurable Feedback Shift Register Cipher Design and Secure Link Layer Protocol for Wireless Sensor Network

by

Guang Zeng
B.Sc., Beijing University of Posts and Telecommunications, 2011

ABSTRACT

Secure wireless communications among sensor nodes is critical to the deployment of wireless sensor networks. However, resource limited sensor nodes cannot afford complex cryptographic algorithms. In this thesis, we propose a low complexity and energy efficient reconfigurable feedback shift register (RFSR) stream cipher, link layer encryption framework RSec and authentication protocol RAuth.

RFSR adds one new dimension, reconfigurable cipher structure, to the existing stream ciphers. The proposed RFSR is implemented on a field programmable gate array platform. Simulation results show that much lower power consumption, delay and transmission overhead are achieved compared to the existing microprocessor based cipher implementations. The RSec framework utilizes RFSR ciphers to guarantee message confidentiality. By comparing with other encryption frameworks in terms of energy efficiency, RSec achieves the best benchmark. The RAuth protocol is designed on top of RFSR and RSec. It provides excellent authentication speed and security level by comparing with other authentication protocols.

Contents

List of Tables

List of Figures

GLOSSARY

ACK	Acknowledgment
ASIC	Application Specific Integrated Circuit
CRC	Cyclic Redundancy Check
FPGA	Field Programmable Gate Array
IV	Initial Vector
LEAP	Localized Encryption and Authentication Protocol
LFSR	Linear Feedback Shift Register
MAC	Message Authentication Code
MIC	Message Integrity Code
NFSR	Non-linear Feedback Shift Register
PER	Packet Error Rate
RAuth	RFSR Cipher based Authentication Protocol
RSec	RFSR Cipher based Secure Sensor Network Communication Architecture
RFSR	Reconfigurable Feedback Shift Register
RSSI	Receive Signal Strength Indicator
SNR	Signal to Noise Ratio
WSN	Wireless Sensor Network
XOR	Exclusive OR

ACKNOWLEDGEMENTS

I would like to thank:

Co-Supervisors Dr. Dong and Dr. Bornemann, for your mentoring, encouragement, and patience.

My Families, for your support and love.

My Colleagues and Friends, for your inspiration and help.

DEDICATION

To my family and friends.

Chapter 1

Introduction

After years of research and development, wireless sensor networks (WSNs) are being deployed for various industrial and consumer applications. The low cost makes them possible to be deployed in a large scale in various markets performing both military and civilian tasks. The tiny sensor nodes, with the abilities of data sensing, data processing and communications, become a more suitable choice in situations where traditional networks are technically hard or expensive to be utilized. However, sensor nodes also suffer from resource constraints due to the limited size and the intention of low cost in the design phase. Sensor nodes in a WSN are usually deployed in an unknown environment which can be hostile. Besides, due to the nature of wireless communications media, any adversary with proper radio modules can overhear the communications in the air. Therefore, secure communication mechanisms should be utilized to protect the confidentiality of the information exchanging on the media. A WSN is special when compared to a traditional computer network. Certain constrains in WSN make it inefficient and sometimes impossible to use the existing network security mechanisms directly. Hence, there is an urge of developing security approaches specifically for a sensor network.

In order to develop the security protocols suitable for sensor networks, it is necessary to know and understand the constraints first [3, 4].

Power limitation is one of the major constraints. A sensor node's life period mostly relies on the battery capacity it carries. In order to make a sensor node work as long as possible, well performing yet power consuming processors are replaced by regular processors with low energy requirements, and radio modules are usually configured with lower transmission power and higher receiving sensitivity to reduce power consumption. Therefore, the designs of a sensor node and a WSN system should always keep energy consumption in mind. As for the security mechanism, the public-key based algorithms used in traditional networks are much too power consuming for a sensor node, let alone the computational overhead which will bring significant delay.

Wireless Communications is another problem to WSN security. A wireless signal can be picked up by an adversary with the similar radio module used in a WSN. Since the WSN cannot afford expensive encryption algorithms or more secure authentication protocols used in traditional networks, a specially designed protocol stack is in great need to protect its confidentiality. Low power wireless communication suffers from packet loss and bit errors, which should be considered carefully and handled efficiently in a protocol. Packet collisions are another issue to take care of. A sensor node cannot utilize strong time synchronization due to its large overhead. Therefore, the media access technique should be carefully designed to decrease the possibility of packet conflicts.

Unattended Situations are common for a WSN when the nodes are left unattendedly to work automatically for long periods of time. When nodes are deployed in a hostile environment, it is quite possible that physical attacks are launched by adversaries.

Despite the constrains mentioned above, a WSN has several critical security requirements.

Data Confidentiality is the most important issue in network security. A sensor node should not leak local data to its neighboring nodes. The common approach for keeping sensitive data secret is to encrypt data with a secret key that only the intended receivers can process, hence achieving confidentiality.

Data Authentication is important for many applications in a WSN. Authentication is necessary for many administrative tasks. Since adversaries also have access to the communication media, they can easily inject illegal messages to the network. This requires all the receivers in a WSN to verify whether the data used in any decision-making process originates from the correct source. Informally, data authentication allows a receiver to verify that the data really was sent by the claimed sender.

Data Integrity guarantees that the received message is the exact copy of the message sent from the sending node. Due to the unreliable nature of the wireless communication environment, traffic collision, bit error, etc., are likely to happen and result in the received packet being useless. An adversary may add some fragments or manipulate the data in a packet and send it to the original destination. Data integrity check can detect malicious behaviors or data damage due to harsh environments.

Data Freshness should be verified even if confidentiality and data integrity are assured. Informally, data freshness implies that the data is recent, and it ensures that no adversary replayed old messages.

1.1 Motivation and Related Work

In recent years, Internet of Things has become a popular topic in both the academic community and the industry. Connecting everything to the Internet is changing from just a slogan to something possible. Sensors and automatic controllers are invading our homes and work spaces. More and more sensors and devices will be connected to the networks, some of which may even perform critical tasks. Therefore, research on the secure communications in a WSN has a growing importance.

The progress in the industry brings new concepts, advanced hardwares and new application markets to WSNs. Some assumptions in previous research changed and some important constraints were overcome for the benefits of hardware improvements. Therefore, it makes perfect sense to design new protocol stacks to meet the up-to-date requirements.

1.1.1 Cryptography Algorithms for a WSN

Cryptography algorithms can be basically classified into two categories, public key cryptography and private key cryptography. The public key cryptography, known as the asymmetric cryptography, requires two keys to perform a cryptography task, a public key and a private key. The two keys are mathematically one-to-one related. The distinguishing technique used in public-key cryptography is that one of the two keys is used to encrypt a message while the other key is used to decrypt it. The private key cryptography, known as the symmetric key cryptography, uses the same private key to encrypt and decrypt a message.

Many researchers believe that it is undesirable to employ public key algorithms on sensor nodes, such as the Diffie-Hellman key agreement protocol [5], RSA signatures [6], and Elliptic Curve Cryptography (ECC) [7]. RSA and ECC are the two major cryptography algorithms in the literature. ECC offers equal security for a far smaller key size and therefore reduces processing and communication overheads. Table 1.1 summarizes the execution times of ECC and RSA implementations on an Atmel ATmega128 processor (used by Mica2 mote) [1]. The execution time is measured on average for a point multiplication in ECC and a modular exponential operation in RSA. Two standardized elliptic curves, ECC secp160r1 and secp224r1, are defined in [8]. As shown in Table 1.1, by using a relatively small integer $e = 2^{16} + 1$ as the public key, RSA public key operation is slightly faster than ECC point multiplication. However, ECC point multiplication outperforms RSA private key operation by an order of magnitude. The RSA private key operation, which is too slow, limits its use in a sensor node. ECC has no such issues since both the public key operation and private key operation use the same point multiplication operations.

Table 1.1: Average operation time of public key cryptography algorithm ECC and RSA [1]

Algorithm	Operation Time (s)
ECC secp160r1	0.81
ECC secp224r1	2.19
RSA-1024 public-key e = $2^{16} + 1$	0.43
RSA-1024 private key w. CRT[1]	10.99
RSA-2048 public-key e = $2^{16} + 1$	1.94s
RSA-2048 private-key w. CRT[1]	83.26

Wander et al. investigated the energy cost and time delay of authentication and key exchange based on RSA and ECC algorithms on the platform with Atmel ATmega128 processor [2]. The WSN is assumed to be centralized, where each sensor node has a certificate signed by the administrator node's private key using a RSA or ECC signature. Elliptic Curve Digital Signature Algorithm (ECDSA) is used to generate and verify the ECC-based signature. With a key exchange protocol similar to SSL 3-way handshake [9], the two parties validate the certificates of each other before a session key used in later communications is negotiated. This research shows that the ECC-based key exchange protocol has a better performance than the RSA-based key exchange protocol at the server side but the performances are almost the same

[1] Chinese Remainder Theory

for the client, the sensor node. The results are shown in Table 1.2.

Table 1.2: Public key cryptography: average energy costs of digital signature and key exchange computations [mJ] [2]

Algorithm	Signature		Key Exchange	
	Sign	Verify	Client	Server
RSA-1024	304	11.9	15.4	304
ECDSA-160	22.82	45.09	22.3	22.3
RSA-2048	2302.7	53.7	57.2	2302.7
ECDSA-224	61.54	121.98	60.4	60.4

The result shows that the public key cryptography algorithm seems not to be a good choice for a WSN because it takes thousands or even millions of multiplication instructions to perform a single security operation [10]. Besides, a microprocessor's public key algorithm efficiency is primarily determined by the number of clock cycles required to perform a multiplication instruction [3]. Since it takes much time to perform encryption and decryption operations in constrained devices, this exposes a vulnerability to DoS attacks [11]. It is found that a simple multiplication function with a 128 bit result takes a microprocessor thousands of nano-joules [3]. By comparison, cryptographic hash functions and symmetric key encryption algorithms consume much less computational power than public key algorithms. For example, on a platform with an MC68328 processor, a 1024-bit block takes 42 mJ using RSA while it only takes a 128 bit AES cipher 0.104 mJ. As for the hardware implementations of public key cryptography algorithms, the delay and energy overheads are still too large for a WSN device [12, 13].

Private key cryptography algorithms have shorter time delay and smaller energy overhead, which make them a better choice for WSNs. Reference [14] evaluates five popular encryption schemes, RC4 [15], RC5 [16], IDEA [15], SHA-1 [17], and MD5 [15, 18] on six different microprocessors ranging in word size from 8 bit Atmel AVR to 16 bit Mitsubishi M16C to 32 bit StrongARM, Xscale. For each algorithm and platform, the execution time and code memory size were measured. The results confirm that the private key algorithm outperforms the public key algorithms for a sensor node.

Two symmetric key algorithms RC5 and TEA [19] were evaluated in [20]. They further evaluated six block ciphers, including RC5, RC6 [21], Rijndael [22], MISTY1 [23], KASUMI [24], and Camellia [25] on IAR Systems MSP430F149 in [26]. Code, data memory and CPU cycles are the benchmark criteria. The evaluation results showed that Rijndael is suitable for high-security and energy-efficiency requirements while MISTY1 is good for storage and energy efficiency. The work in [26] provides a good resource for deciding which symmetric algorithm should be adopted in sensor networks.

1.1.2 WSN Key Management and Authentication

Key management is the deterministic factor to ensure WSN security, which helps establish required keys shared between sensor nodes. Since public key cryptography suffers from power and computational limits on WSN platforms, the most proposed key management protocols are based on private key encryption. Based on the probability of key sharing between a pair of sensor nodes, the protocols can be divided into probabilistic key schemes and deterministic key schemes. Based on the network topology, the protocols can be divided into centralized key schemes and distributed key schemes.

In the centralized key management schemes, the central node is the logic center of the network. It controls the key generation and distribution for the WSN. In the distributed, or decentralized key management schemes, two sensor nodes authorize each other without the help of the central node. The nodes will establish pairwise keys with their neighbors simultaneously. Deterministic and probabilistic schemes fall into this category.

In [27], Eschenauer and Gligor introduced a key predistribution scheme for sensor networks which relies on probabilistic key sharing among the nodes of a random graph. Three phases are included in this scheme: key predistribution, shared-key discovery, and path key establishment. In the key predistribution phase, each sensor node keeps a key ring in the memory. In the key ring, k keys are randomly chosen from a key pool of P keys. The base station saves a copy of the association information of the key identifiers in the key ring and the sensor identifier. Each sensor is assumed to share a pairwise key with the base station. In the shared key discovery phase, each sensor looks for his neighbors that they have shared keys within the radio range. In the path-key establishment phase, a path-key is assigned for the nodes that do not share a key but are connected by multiple links at the end of the second phase. Inspired by the work of [27], more random key predistribution schemes have been proposed in [28–33].

On the other hand, the deterministic approaches require that each node is predistributed with the credential information. In the authentication phase, nodes verify their neighbors' identities using the pre-loaded credentials. Zhu et al. proposed the Localized Encryption and Authentication Protocol (LEAP) in [34], which supports the establishment of four types of keys for each sensor node: a predistributed individual key shared with the base station; a predistributed group key shared by all the nodes in the network; pairwise keys shared with direct neighbor nodes; a cluster key shared within a subnetwork. The pairwise keys shared with direct neighbors are used for unicast messages while the cluster key is used for subnetwork broadcast.

In the predistribution phase, each sensor node is loaded with an initial key K_I. The node N calculates its master key by $K_N = f_{K_I}(ID_N)$, where f is a pseudorandom function. In the next phase, neighbor discovery phase, N broadcasts a HELLO message with ID_N. If a neighbor node M receives the broadcast message, it'll reply with $ID_M, MAC(K_M, N|M)$. Node N calculates the pairwise key $K_{NM} = f_{K_M}(N)$. Node M calculates K_{NM} in the same way. Then, the pairwise key is established. Cluster key will be established right after the pairwise key establishment. If node N requires a cluster key, it will generate a random key K_C^N. The cluster key will be sent to each of N's neighbors encrypted by the pairwise key. After a certain time since node deployment, the timers inside sensor nodes will expire and the initial keys are deleted. The author assumed that within the time period, the adversary was not able to compromise a sensor node to get the initial key.

1.2 Contributions

The main contributions of this thesis are summarized as follows.

First of all, a light-weight hardware-oriented cipher is proposed. Almost all the current sensor nodes are using the embedded processor to do the cryptography work while the processor is already largely occupied by other tasks, such as data sensing and processing, interrupt handling, communication protocol processing, etc. The proposed cipher takes over the burden of cryptography tasks so that the node processor will not be overloaded with tasks. Besides, the proposed hardware-based cipher has a dynamic structure which means that the cryptography algorithm is also changeable. The new secrecy dimension makes the cipher much harder to attack. Implemented in hardware, the new cipher has a much smaller delay and a much lower average power consumption which improves the performance by a decent amount. This research work was published in IEEE Wireless Communications Letters [35].

Secondly, a secure centralized authentication protocol is proposed and a link layer encryption protocol is designed using the proposed cipher. The new authentication protocol is more suitable for the situation, where sensor nodes are densely deployed, while it maintains the ability to work similar to a distributed WSN.

1.3 Thesis Outline

The rest of this thesis is organized as follows:

Chapter 2 proposes a hardware-oriented light-weight cipher. Then the security of the new cipher is analyzed. Finally, the cipher is implemented and the performance is simulated.

Chapter 3 introduces a centralized authentication protocol and a link layer encryption protocol. Then the performances of both protocols are analyzed and compared with the existing ones.

Chapter 4 concludes the thesis and suggests possible future work.

Chapter 2

Reconfigurable Feedback Shift Register Based Cipher

Sensor nodes are low-cost, computational- and energy-limited devices which cannot afford resource consuming cryptography algorithms. The fact that anyone with proper receiving tools has access to the signal in the air makes security a main issue of WSNs. Modern WSNs are bi-directional, also enabling sensor nodes to control other logically connected devices. The use of control functions requires higher security mechanisms to prevent attacks. Proper security schemes befitting the requirements of WSN should guarantee both sufficient level of security and low resource consumption.

Conventional public-key cryptography seems feasible but the computational overhead is too large for resource-limited sensor nodes [36]. Private-key cryptography, also known as symmetric cryptography, is suitable for environment constrained applications such as sensor nodes. Private-key encryption uses either stream ciphers or block ciphers. Compared with block ciphers, stream ciphers are often simpler but sufficiently secure. In a stream cipher, plaintext and keystream are bitwise combined using exclusive-or (xor) operation to generate ciphertext. The keystream is a pseudorandom bit stream generated serially using shift registers in a stream cipher. Ciphertext is transmitted over the air between two communicating nodes. The decryption process on a receiving node resembles the encryption process by bitwise xor of the ciphertext with the keystream to restore the plaintext.

Software implementation of cryptography algorithms is usually carried out by the embedded processor in a sensor node. However, the computational resource limited embedded processor is also responsible for other operations such as sensor control, communication protocol execution and sensor data processing. While simple security algorithms may have a weakness for certain security attacks, complex security algorithms will definitely take up much of a processor's resources and negatively impact other real-time tasks running. Hardware encryption implementation frees a processor from heavy duty security function processing and becomes a natural choice for commercial uses such as A5/1 cipher of GSM, E0 cipher of Bluetooth and etc. Hardware oriented stream cipher design has relatively low power consumption, constant and predictable delay and high throughput rate, which makes it a good choice for sensor nodes.

A feedback shift register (FSR) based stream cipher uses feedback update functions to generate new internal states from the current internal states. The feedback update functions are fixed in stream ciphers. Traditional stream ciphers can increase their resistance against attacks by increasing the key and initial vector (IV) sizes. However, if the feedback update functions are designed to be dynamic, attacks will become harder to accomplish because both the cipher structure (the feedback update functions) and the secret key are unknown.

In this chapter, we propose a light-weight hardware-oriented cryptography algorithm, i.e., the reconfigurable feedback shift register (RFSR) based stream cipher, and implement it on a reconfigurable device to test its performance. In our design, the feedback shift register based cipher is structure reconfigurable. This scheme guarantees high message confidentiality for WSNs. Comparing with the existing microprocessor based platforms, the proposed scheme achieves over 130 times less average energy consumption and over 25 times less delay.

The remainder of the chapter is organized as follows. Section 2.1 introduces the system model under consideration. In Section 2.2 the RFSR based cipher is proposed. The security of RSFR cipher is analyzed in Section 2.3. Implementation, simulation and performance is detailed in Section 2.4, and Section 2.5 concludes this chapter.

2.1 System Model

The network model and the security model are analyzed in this section. The network model describes the network topology, i.e., how the sensor nodes are organized, the schemes of data encryption and transmission in different layers, and the secret key deployment and management mechanisms. The security model indicates the potential attacks a sensor node may encounter in this chapter.

2.1.1 Network Model

A wireless sensor network is composed of resource limited sensor nodes, power sufficient sink nodes and a base station. A WSN contains one base station (BS) and several sink nodes depending on the network topology. A sensor node communicates with a BS in one hop or multiple hops through sink nodes. Three abstract layers are considered on the device within the WSN: the physical layer, the link layer and the application layer. The physical layer is a fundamental layer, consisting of the basic networking hardware transmission technologies. The link layer is the protocol layer that transfers data between sensor nodes in the WSN. The application layer is responsible to handle system and node specific tasks. Due to the insecure nature of WSN, encryption is performed in both the application layer and the data link layer. The application layer encryption guarantees that only the destination and the original sender have access to the data. The link layer encryption requires unique pairwise ciphers established between neighboring nodes to protect message transmission in a direct link. Each sensor node is equipped with both a microprocessor and a hardware component. The hardware components can be reconfigurable devices such as field-programmable gate arrays (FPGAs) or application-specific integrated circuits (ASICs). The hardware component can be used for handling security related tasks. In this chapter, we simulate FPGA implementations to demonstrate the performance of the proposed cipher. In the simulation, the hardware component is used only for encryption and decryption.

2.1.2 Security Model

We assume that sensor nodes have no tamper-resistant mechanism. Once sensor nodes are captured and compromised by an adversary, all stored data such as cipher structures and keys will be exposed in a short time and can be utilized by reprogramming the captured nodes. An adversary can also launch passive attacks which attempt to break the cipher by eavesdropping on communications between legitimate nodes. Denial of service (DoS) attacks can be mounted to disrupt regular communications between nodes, or to drain up nodes' energy. DoS attacks in the data link layer and the application layer are considered in the chapter.

2.2 The RFSR Cipher

Symmetric cryptographic primitives for encryption are divided into block ciphers and stream ciphers. Block ciphers operate on fixed-length groups of bit blocks. Substitutions and permutations are two simple operations to effectively improve the security in block ciphers. Stream ciphers work in different ways from block ciphers: they maintain a secret state which changes with time during the encryption; they produce bit streams rather than bit blocks in block ciphers. Therefore, the two characters of stream ciphers are: a state-update function, which generates the new cipher state based on the previous cipher state, and an output function, which produces the output by filtering the cipher state. The output of a stream cipher is XORed with the plaintext to get the ciphertext. The stream ciphers are similar to a one-time pad (OTP) cipher. Without the long secret key in OTP, stream ciphers use a secret key to generate pseudo-random bit streams, which is computationally indistinguishable from a stream of random bits.

The proposed RSFR cipher is a stream cipher, which is partially based on the design of the Grain cipher [37]. Therefore we will first briefly review the Grain cipher and its application in WSN in Section 2.2.1. Afterwards, the detailed design of the RFSR cipher is described in Section 2.2.2. The initialization process of the Grain and RFSR ciphers is presented in Section 2.2.3. Cipher management, IV management follows in Section 2.2.4 and 2.2.5. Finally, the key and structure update scheme is introduced in Section 2.2.6.

2.2.1 Grain Cipher

Grain ciphers are a family of stream ciphers selected in the final portfolio of Profile 2 (for hardware applications) in the eSTREAM project [38]. It is known for its hardware-oriented, elegant and simple design. The first version of the Grain cipher is targeting on applications which require low hardware complexity, such as radio frequency identifications(RFIDs) and WSN nodes.

The design is based on two shift registers, one with a linear feedback shift register (LFSR) and one with a non-linear feedback shift register (NFSR). The state-update functions, in this case the linear and non-linear feedback functions, are carefully designed and hard coded. The LFSR guarantees a minimum period for the keystream and it also provides balance in the output. The NFSR, together with a nonlinear output function, introdues nonlinearity to the Grain cipher. The state-change input to the NFSR is masked with the output of the LFSR states so that the state of the NFSR is also balanced. Keys, IVs and padding bits are used as the initial values of the cipher internal state. The original design of Grain uses 80-bit keys and 64-bit IVs. The new version Grain 128 [39] has 128-bit keys and 96-bit IVs.

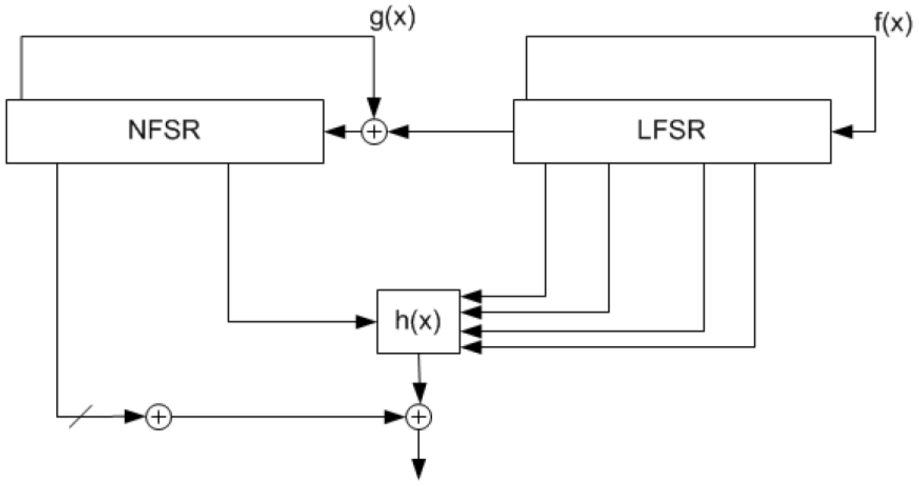

Figure 2.1: Grain cipher version 1 structure

In Grain cipher version 1, which is shown in Fig. 2.1 , the content of the LFSR is denoted by $s_i, s_{i+1}, ..., s_{i+79}$ and the content of the NFSR is denoted by $b_i, b_{i+1}, ..., b_{i+79}$. The feedback polynomial of the LFSR, $f(x)$ is a primitive polynomial of degree 80. It is defined as

$$f(x) = 1 + x^{18} + x^{29} + x^{42} + x^{57} + x^{67} + x^{80}.$$

The above function is expressed in finite field arithmetic as a polynomial mod 2, which differs from the integer arithmetic. This means that the coefficients of the polynomial mush be 1's or 0's.

To remove any possible ambiguity, the update function of the LFSR is defined as

$$s_{i+80} = s_{i+62} + s_{i+51} + s_{i+38} + s_{i+23} + s_{i+13} + s_i$$

The feedback polynomial of the NFSR, $g(x)$, is defined as

$$g(x) = 1 + x^{18} + x^{20} + x^{28} + x^{35} + x^{43} + x^{47} + x^{52} + x^{59} + x^{66} + x^{71} + x^{80} + x^{17}x^{20} + \\ x^{43}x^{47} + x^{65}x^{71} + x^{20}x^{28}x^{35} + x^{47}x^{52}x^{59} + x^{17}x^{35}x^{52}x^{71} + x^{20}x^{28}x^{43}x^{47} + \\ x^{17}x^{20}x^{59}x^{65} + x^{17}x^{20}x^{28}x^{35}x^{43} + x^{47}x^{52}x^{59}x^{65}x^{71} + x^{28}x^{35}x^{43}x^{47}x^{52}x^{59}$$

In the same way, to remove any possible ambiguity, we also write the update function of the NFSR as

$$\begin{aligned} b_{i+80} = {} & s_i + b_{i+62} + b_{i+60} + b_{i+52} + b_{i+45} + b_{i+37} + b_{i+33} + b_{i+28} + b_{i+21} + b_{i+14} + b_{i+9} + \\ & b_i + b_{i+63}b_{i+60} + b_{i+37}b_{i+33} + b_{i+15}b_{i+9} + b_{i+60}b_{i+52}b_{i+45} + b_{i+33}b_{i+28}b_{i+21} + \\ & b_{i+63}b_{i+45}b_{i+28}b_{i+9} + b_{i+60}b_{i+52}b_{i+37}b_{i+33} + b_{i+63}b_{i+60}b_{i+21}b_{i+15} + \\ & b_{i+63}b_{i+60}b_{i+52}b_{i+45}b_{i+37} + b_{i+33}b_{i+28}b_{i+21}b_{i+15}b_{i+9} + b_{i+52}b_{i+45}b_{i+37}b_{i+33}b_{i+28}b_{i+21} \end{aligned}$$

Note that the bit s_i, which is from the LFSR internal state, is masked with the input in the NFSR update function.

The two shift registers together form the internal state of the Grain cipher. The two update functions determine the next state based on the current state. From the internal state, five bits are taken as input to a boolean function, $h(x)$. This output function is chosen to be balanced, correlation immune of the first order and has algebraic degree 3. The nonlinearity is the highest possible for these functions, namely 12. The input is taken both from the LFSR and from the NFSR. The function is defined as

$$h(x) = x^1 + x^4 + x^0x^3 + x^2x^3 + x^3x^4 + x^0x^1x^2 + x^0x^2x^3 + x^0x^2x^4 + x^1x^2x^4 + x^2x^3x^4$$

where the variables x^0, x^1, x^2, x^3 and x^4 correspond to the tap positions s_{i+3}, s_{i+25}, s_{i+46}, s_{i+64}, b_{i+63} respectively. The output function is taken as

$$z_i = b_{i+1} + b_{i+2} + b_{i+4} + b_{i+10} + b_{i+31} + b_{i+43} + b_{i+56} + h(s_{i+3}, s_{i+25}, s_{i+46}, s_{i+64}, b_{i+63})$$

Research in time-memory-data trade-off attacks suggests that it is possible to mount an attack with complexity $O(2^{K/2})$ where K is the size of the key. In this scenario, the attacker has a collection of $2^{K/2}$ plaintexts encrypted under different keys, and the aim of the attack is to find one of these keys. In this attack scenario, 80 bit key size is not enough since an attack would have complexity $O(2^{40})$. Several researchers have expressed the opinion that 128 bit keys is a minimum in secure applications.

To meet this new requirement, Grain-128 cipher, which is drawn in Fig. 2.2, was proposed while preserving the advantages of Grain cipher version 1. It uses 128-bit key and 96 bit IV. Similarly, the cipher consists of three main building blocks, namely an LFSR, an NFSR and an output function. The content of the LFSR is denoted by $s_i, s_{i+1}, ..., s_{i+127}$ and the content of the NFSR is denoted by $b_i, b_{i+1}, ..., b_{i+127}$. The feedback polynomial of the LFSR is a primitive polynomial of degree 128, which is defined as

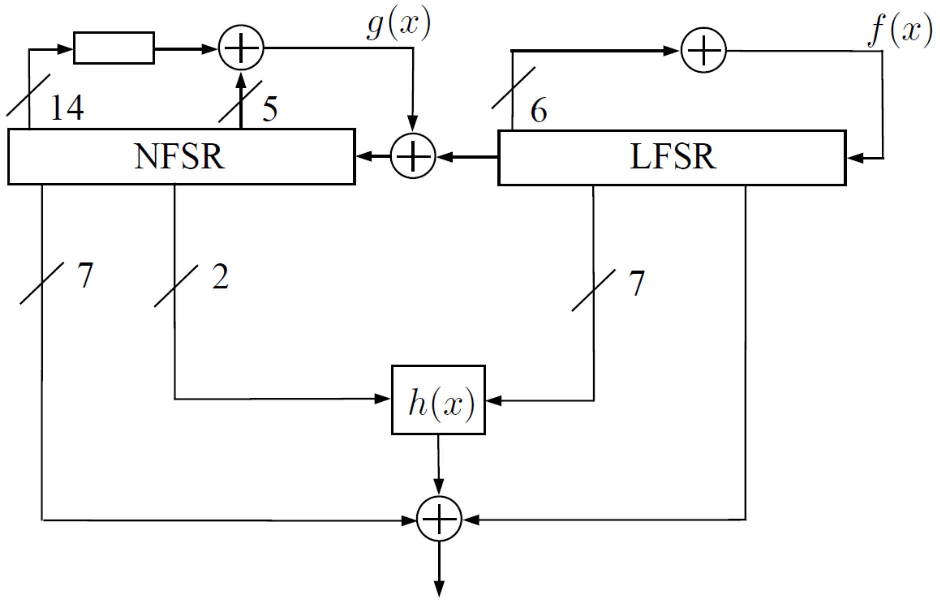

Figure 2.2: Grain-128 cipher structure

$$f(x) = 1 + x^{32} + x^{47} + x^{58} + x^{90} + x^{121} + x^{128}$$

The corresponding update function of the LFSR is

$$s_{i+128} = s_{i+96} + s_{i+81} + s_{i+70} + s_{i+38} + s_{i+7} + s_i$$

The non-linear feedback polynomial of the NFSR, $g(x)$, is the sum of one linear element and non-linear elements, which is defined as

$$g(x) = 1 + x^{32} + x^{37} + x^{72} + x^{102} + x^{128} + x^{44}x^{60} + x^{61}x^{125} + x^{63}x^{67} + x^{69}x^{101} + x^{80}x^{88} + x^{110}x^{111} + x^{115}x^{117}$$

Similarly, the bit s_i, which is masked with the input to the NFSR, is included while omitted in the feedback polynomial. The corresponding update function of the NFSR is defined by

$$b_{i+128} = s_i + b_{i+96} + b_{i+91} + b_{i+56} + b_{i+26} + b_i + b_{i+84}b_{i+68} + b_{i+65}b_{i+61} + b_{i+48}b_{i+40} + b_{i+59}b_{i+27} + b_{i+18}b_{i+17} + b_{i+13}b_{i+11} + b_{i+67}b_{i+3}$$

The 256 memory elements in the two shift registers represent the state of the cipher. From this state, 9 variables are taken as input to a Boolean function, $h(x)$. Two inputs to $h(x)$ are taken from the NFSR and seven are taken from the LFSR. This function is of degree 3 and very simple. It is defined as

$$h(x) = x^0 + x^1 + x^2x^3 + x^4x^5 + x^6x^7 + x^0x^4x^8$$

where the variables x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7 and x_8 correspond to the tap positions b_{i+12}, s_{i+8}, s_{i+13}, s_{i+20}, b_{i+95}, s_{i+42}, s_{i+60}, s_{i+79} and s_{i+95}, respectively. The output function is defined as

$$z_i = b_{i+2} + b_{i+15} + b_{i+36} + b_{i+45} + b_{i+64} + b_{i+73} + b_{i+89} + \\ h(b_{i+12}, s_{i+8}, s_{i+13}, s_{i+20}, b_{i+95}, s_{i+42}, s_{i+60}, s_{i+79}, s_{i+95}) + s_{i+93}$$

How the IVs are managed and used is not taken into consideration in the Grain cipher design. Actually, the IVs are transmitted in clear text form without encryption for the convenience of the receiver to synchronize and decrypt the received messages. Previous research shows that radio transmission consumes much more power than cryptographic algorithm computation on a variety of sensor nodes [2] [40]. Therefore, to achieve low power consumption, we need to reduce the unnecessary overheads. Since IV is transmitted in each packet, reducing the size of the IV will significantly decrease transmission energy consumption. Besides, long bit length keys will lead to large communication overheads on key establishment and update process. However, smaller key and IV sizes will decrease the security level of the cipher.

With all the above concerns in mind, we try to redesign a cipher not only with smaller key and IV sizes to reduce the transmission overheads, but also with the competitive or even higher security level compared to the Grain cipher. We achieve the goal with a sufferable increase of the hardware complexity. Since the structure, i.e. the feedback update functions and the output function, of the Grain cipher is fixed, we intend to make it reconfigurable in order to bring in randomness which adds to the cipher structure another dimension of the cipher secrecy along with the secret key.

2.2.2 RFSR Cipher

Similar to the Grain cipher, the proposed reconfigurable feedback shift register based cipher, depicted in Fig. 2.3, consists of three main building blocks, namely the LFSR with linear feedback update function f, the NFSR with non-linear feedback update function g, and the output function h. In our design, we use a 32-bit LFSR and a 64-bit NFSR. Other choices of sizes can be carefully designed to fit specific security requirements. The states of the LFSR are denoted as $y_1, y_2, ..., y_{32}$. Similarly, the states of the NFSR are denoted as $z_1, z_2, ..., z_{64}$. The reconfigurable feedback update function of the LFSR, f, is defined as

$$f : y_0 = y_{a_1} + y_{a_2} + y_{a_3} + y_{a_4} + y_{a_5} + y_{32}$$

where a_1, a_2, a_3, a_4 and a_5 are carefully chosen so that the update feedback function f is a primitive polynomial of degree 32. Function f, being a primitive polynomial, guarantees that the internal states of the LFSR can reach the maximum period $2^n - 1$ as long as the initial state is not all zero bits, where n is the bit length of the internal states of the LFSR. Since the primitive polynomial has been studied extensively, taps a_1, a_2, a_3, a_4 and a_5 of the LFSR feedback update function in our design are randomly chosen from an existing structure pool, containing 5039 primitive polynomials of degree 32 [41].

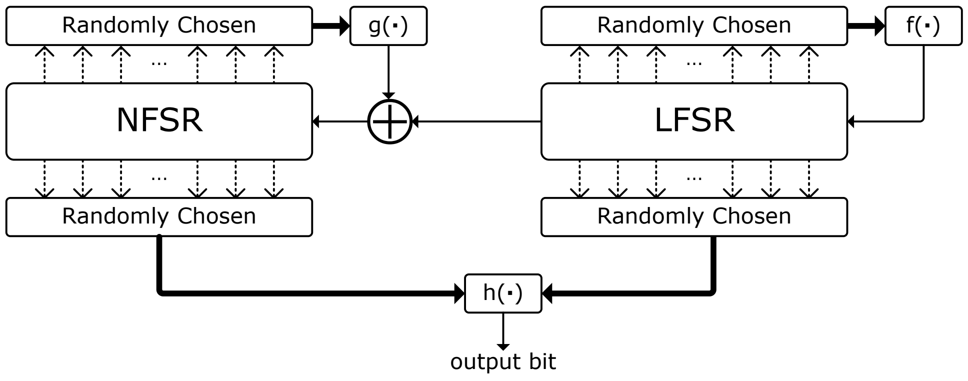

Figure 2.3: RFSR cipher structure

The feedback update function of the NFSR, g, is denoted by

$$g : z_0 = y_{32} + z_{64} + z_{b_1} + z_{b_2} + z_{b_3} + z_{b_4} + z_{b_5} \cdot z_{b_6} + z_{b_7} \cdot z_{b_8} + z_{b_9} \cdot z_{b_{10}} + z_{b_{11}} \cdot z_{b_{12}} + z_{b_{13}} \cdot z_{b_{14}} \cdot z_{b_{15}} + z_{b_{16}} \cdot z_{b_{17}} \cdot z_{b_{18}} \cdot z_{b_{19}}$$

where z_{b_1} to $z_{b_{19}}$ are randomly but not repeatedly chosen from the states of the NFSR, z_1 to z_{63}. According to boolean algebra, repeating values in b_1 to b_{19} will reduce monomial numbers of the polynomial and then compromise the intended security level. Therefore, no repeat values are accepted.

The output function h gets the input from the states of both LFSR and NFSR. It is defined as

$$h : output = y_{c_1} + z_{d_1} + z_{d_2} + z_{d_3} + y_{c_2} \cdot y_{c_3} + y_{c_4} \cdot z_{d_4} + z_{d_5} \cdot z_{d_6} + y_{c_5} \cdot z_{d_7} \cdot z_{d_8}$$

where y_{c_1} to y_{c_5} and z_{d_1} to z_{d_8} are randomly but not repeatedly chosen from y_1 to y_{32} and z_1 to z_{64}, respectively.

For the RFSR cipher, the feedback functions f,g and h are all reconfigurable while these functions in Grain 128 are fixed. In the RFSR cipher, f is composed of 4 or 6 dynamic taps while the linear feedback update function in Grain 128 has 6 fixed taps; g is composed of 6 degree-one, 4 degree-two, 1 degree-three and 1 degree-four monomials while that of Grain 128 is composed of 6 degree-one, 7 degree-two monomials; h is composed of 4 degree-one, 3 degree-two and 1 degree-three monomials while that of Grain 128 is composed of 8 degree-one, 4 degree-two and 1 degree-three monomials.

2.2.3 Cipher Initialization

The cipher will firstly be initialized with the key and the IV before the keystream is generated. Grain cipher version 1 uses 80-bit key and 64-bit IV. The bits of the key is denoted as $k_i, 0 \leq i \leq 79$ and the bits of the IV is denoted as $IV_i, 0 \leq i \leq 63$. The key is loaded in NFSR, where $b_i = k_i, 0 \leq i \leq 79$, and the IV is loaded in LFSR, where $s_i = IV_i, 0 \leq i \leq 63$. The remaining LFSR bits are loaded with 1s, $s_i = 1, 64 \leq i \leq 79$. Since the padding 1s in LFSR, the cipher won't be initialized to the all zero state. The initialization process requires the clock clocked 160 times without producing any keystream. Therefore, during the initialization, the output bit is fed back and XORed on the feedback bits s_{i+79} and b_{i+79} of the linear and non-linear feedback functions, shown in Fig. 2.4.

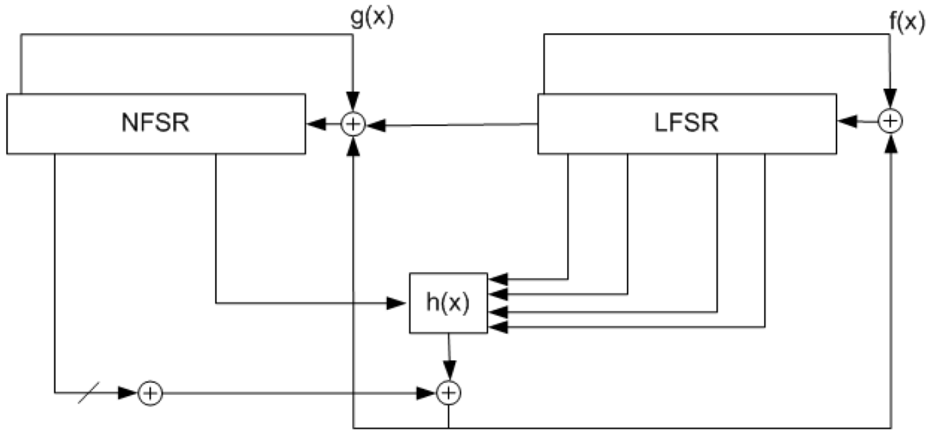

Figure 2.4: Grain cipher version 1 key initialization

For the Grain-128 cipher, the process is generally the same. The key is loaded in NFSR, where $b_i = k_i, 0 \leq i \leq 127$, and the IV is loaded in LFSR, where $s_i = IV_i, 0 \leq i \leq 95$. The last 32 bits of the LFSR is filled with 1s, $s_i = 1, 96 \leq i \leq 127$. The cipher is clocked 256 times before generating keystream to finish the initialization process, shown in Fig. 2.5.

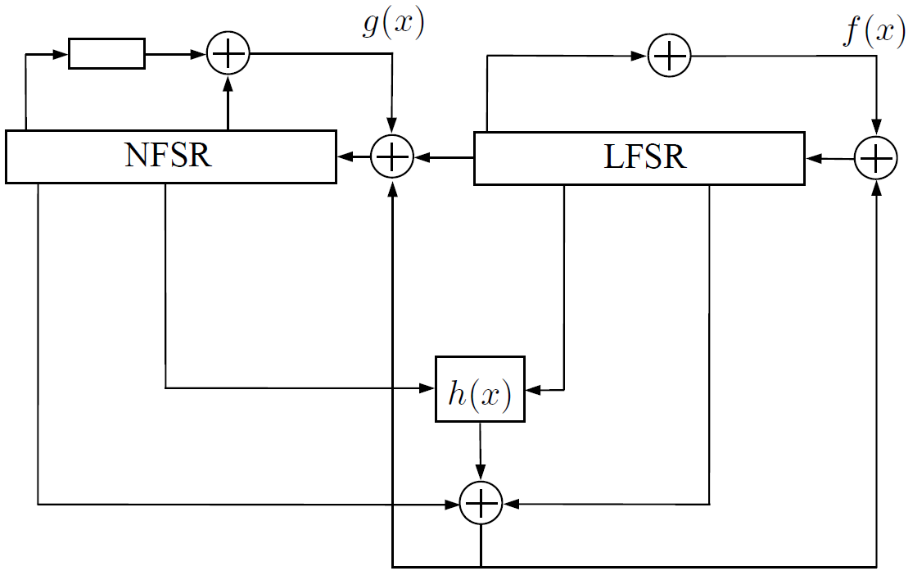

Figure 2.5: Grain-128 cipher key initialization

Stream ciphers need the initialization process before keystream generation due to the randomization requirements. The RFSR cipher initialization process, shown in Fig. 2.6, is executed whenever the cipher is loaded with a new key-IV pair. The output bits are fed back to XOR with the bits calculated by feedback functions f and g. For the cipher designed in this chapter, it is first clocked 96 times without producing the keystream. 96 is the sum of the lengths of LFSR and NFSR. 96 clocks make sure that all the bits of the cipher initial state have influence on the cipher state after initialization.

2.2.4 Cipher Management

In a WSN, one sensor node may need to communicate with several nodes. The encryption may also be utilized in different network layers. So one sensor node need to be able to use multiple ciphers to satisfy the requirements. The RFSR ciphers share the basic hardware structure and differ in the key, feedback and output functions.

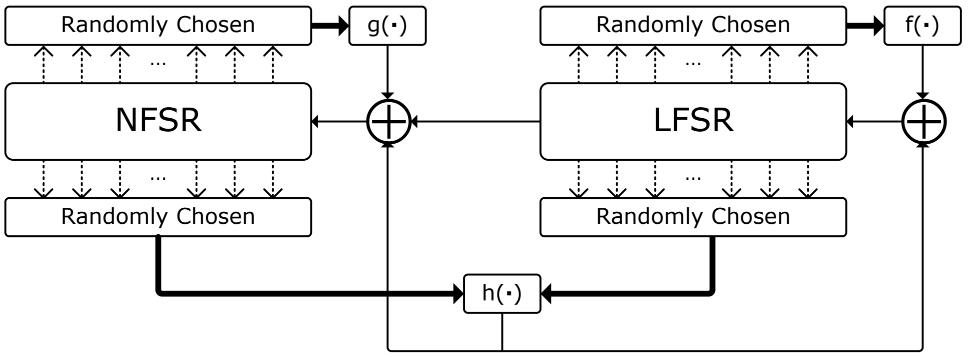

Figure 2.6: RFSR initialization

Therefore, one RFSR cipher hardware implementation is sufficient for a sensor node. The cipher information about key and functions can be saved in storage.

To manage multiple cipher information, we need to store all cipher information and load the particular cipher information to hardware upon requirement. For each cipher, the key, the IV and the taps of the feedback functions and output function should be stored. The key and the IV are the internal states of the cipher which is 96 bits. For the 32-bit linear feedback function, 4 or 6 taps are used. Since y_{32} is used in all polynomials, we only store the remaining 3 or 5 taps which are indexes ranging from 1 to 31. To make the storage neat, we choose to store 5 taps each with 5 binary bits for all linear feedback functions. Therefore, 25 bits are used for linear feedback functions. For the 64-bit non-linear feedback function, 19 taps ranging from 1 to 63 are reconfigurable. Each tap needs 6 bits which is 114 bits in total for the non-linear feedback taps storage. For the taps of the output function, 5 taps from the LFSR states and 8 taps from the NFSR states are used which requires 73 bits in total.

As described above, for one cipher, we need 308 bits for storage in total. When one cipher is needed, the system will load the stored cipher information into the hardware. When the use of one cipher is finished, only the changed elements will be written back to the storage. For example, after continuously generating keystream, the internal states of the cipher are changed while the cipher structure remains unchanged. In this case, only the internal states need to be written back to the storage.

2.2.5 IV Management

As one part of the cipher's initial state, the IV is crucial to the modern stream cipher because its randomness makes the cipher's initial states different in each use which finally results in a different keystream for each piece of message to be transmitted. The IV is combined with the secret key together as the cipher's initial states. IV should never repeat with the same key. If so, the keystream will be identical which will leak unnecessary information of the plaintext. Suppose plaintext $pt1$ and $pt2$ are encrypted with the same key k and initial vector IV while the k and IV combination will produce the keystream $keystream$. The encrypted message of $pt1$ and $pt2$ will be ciphertext $ct1$ and $ct2$, respectively. We get

$$\begin{aligned} keystream &= Cipher(k, IV) \\ ct1 &= pt1 \oplus keystream \\ ct2 &= pt2 \oplus keystream \end{aligned}$$

As ciphertext $ct1$ and $ct2$ are transmitted in the air, they are also exposed to the adversary. $ct1$ and $ct2$ can reveal part of the plaintext information by

$$ct1 \oplus ct2 = pt1 \oplus pt2$$

Therefore, the IV should never repeat with the same key.

The IV can be used in two methods: the whole IV method and the IV index method. In the whole IV method, the IV is transmitted in clear text in each packet; while in the IV index method, only the index of the IV is transmitted instead. Obviously, the index of the IV can be much smaller than the IV itself. Then the IV index method requires less bits for transmission in each packet than the whole IV method. Another advantage of the IV index method in our scheme is that the use of IV index will not reveal any part of a keystream. The detailed usage of IV is discussed in the next chapter.

2.2.6 Key and Structure Update Scheme

As a common security mechanism, the key update process is carried out to guarantee that the key in use is safe and secure. The two parties in a communication link will negotiate and perform the key update process depending on the specific protocol in use.

In addition to the key update of traditional stream ciphers, the proposed RFSR cipher can also update its structure. The structure update consists of three basic elements: f update, g update, and h update. A system can carry out partial or total structure updates which means that one or two, or all of the three basic elements are updated. Any change of the cipher structure will completely change the output keystream and result in a brand new cipher. Structure update can use the keystream generated by the RFSR cipher as the source of taps generator for feedback functions. Proper algorithms can be designed to update cipher structures by using the keystream.

Since the linear feedback function f is chosen from an existing structure pool, the structure update of f cannot be generated by the keystream. But the non-linear feedback update function g and the output function h can be generated based on the keystream. The following is an example of a structure update generator algorithm.

Algorithm Random Taps Chosen Algorithm

```
while $TapsArray$ NOT FULL do
    $NewTap \leftarrow$ bits from $Keystream$
    if $NewTap$ NOT IN $TapsArray$ then
        $NewTap$ added to $TapsArray$
    end if
end while
```

If the algorithm is shared by all the sensor nodes in a WSN, when two nodes on the same communication link decide to perform a structure update, they do not have to transmit new structure information. Since they share the same random bit stream source, which is the keystream, and the same structure update algorithm, they can generate the same structures for update purpose.

2.3 Security Analysis

2.3.1 Cipher Security

Since the proposed RFSR cipher is designed based on the Grain cipher, the cryptographic analysis on Grain can also be applied to RFSR ciphers. By now, no key recovery attacks better than brute force attacks are known against Grain 128, indicating the level of security of Grain. Several minor differences between RFSR and Grain are analyzed below.

Compared with Grain 128, the simplified feedback and output functions and the smaller 96 internal states of 32-bit LFSR and 64-bit NFSR, rather than 256 internal states of 128-bit LFSR and NFSR in Grain 128, seem to make the RFSR cipher more vulnerable to attacks. However, the changeable cipher structure makes the RFSR cipher much more difficult to succumb to attacks. Assume an adversary may have access to the entire 5039 LFSR structure pool by compromising a large number of sensor nodes. The basic NFSR feedback boolean function and basic output boolean function may also be available. But the random taps used in the NFSR feedback function or in the output function are still unknown, which are 19 taps of NFSR states in the NFSR feedback function, 8 taps of NFSR states and 5 taps of LFSR states in the output function. The total possible structure will be $5039 * \binom{63}{19} * \binom{32}{5} * \binom{64}{8} \approx 1.33 * 2^{114}$. Therefore, it is hard to launch attacks on RFSR ciphers.

2.3.2 Attack Analysis

The large number of possible cipher structures makes eavesdropping hard to compromise system security. Note that different pairwise RFSR ciphers are established and used in the data link layer between neighboring sensor nodes, and another RFSR cipher is used in the application layer between the BS and the source sensor node. Even when several nodes are captured by adversaries, only the ciphers owned by these nodes are exposed but they cannot be utilized to break uncompromised nodes' ciphers. DoS attacks and forgery from an outsider are defended with the use of a message authentication code (MAC). A MAC is added to each message's payload and helps the receiver verify the authenticity and integrity of the received messages. A message is considered valid only if the received MAC is correct. The remedies for the DoS attacks coming from the captured nodes have been proposed in the literature, such as switching to low duty cycle and conserving power, locating attack area and re-routing traffic, and adopting prioritized transmission [42].

2.4 Implementation, Simulation and Performance

Altera Cyclone II EP2C8T144C6 FPGA was chosen as the target implementation device. The simulation software platforms are Altera Quartus II V12.1 and Mentor Graphics ModelSim SE 10.1a. We simulate the power consumption using the Altera PowerPlay power analysis tool [43]. PowerPlay uses actual design placement and routing and logic configuration which is claimed to be accurate (to within ±10%) for the actual device power consumption [44]. Existing experiments [45] also show that the result of PowerPlay power estimation on the Cyclone II series is reasonable.

The total FPGA power consumption comprises static power and dynamic power. Static power is the power consumed by a device due to leakage currents when in quiescent state. Dynamic power is the additional power consumed through device operation caused by signals toggling and capacitive loads charging and discharging. Therefore, with increasing operating clock frequency, the dynamic power increases accordingly but the static power remains the same.

Firstly we execute gate-level timing simulation, which takes all the routing resources and the exact logic array resource usage into account to obtain an accurate power estimation. Then PowerPlay is run to measure the average power consumption of each operation. We obtain the power consumption directly from the PowerPlay tool report and calculate the energy-per-bit performance.

2.4.1 Cipher Implementation

The proposed implementation achieves several cipher functionalities with only one structure implementation. Each cipher's specific information, such as key, IV, feedback taps, and output function, are stored in random access memory (RAM). Since one sensor node needs several RFSR ciphers for data link layer pairwise encryption and application layer encryption, the proposed implementation builds upon a basic cipher infrastructure, and the system automatically loads specific cipher information from RAM when required.

Similar to the Grain's structure, the throughput of the proposed RFSR cipher can be easily multiplied by implementing feedback functions and output functions several times. Average power consumptions are compared with different throughput rates at 1 and 8 bits per clock cycle, and different clock rates at 10 and 50 MHz. We find from simulation that the average energy consumption of 8 bits per clock implementation is almost 6 times less than 1 bit per clock. As expected, with different clock rates, the static power is almost the same but the dynamic power is proportional to the operating frequency.

2.4.2 Comparison with Microprocessor Platforms

Previous research [14] studied the performances of several ciphers and hash functions on microprocessor platforms. We choose ATmega103 and StrongARM microprocessor platforms which respectively represent low-end and high-end processors. Since a microprocessor processes one instruction per clock, the fastest encryption scheme for a particular platform is also the most energy efficient scheme. The most energy efficient algorithms for ATmega103 and StrongARM platforms are RC4 and RC5, respectively. Existing flaws [46] [47] make RC4 and RC5 susceptible to attacks, while the brute force attack remains one of the most effective attack against Grain 128, indicating the higher security of Grain 128. The plaintext to be encrypted is 512 bits long, and initialization is executed before encryption. Per bit energy consumption is calculated by averaging both initialization and encryption energy consumption over the total 512 bits.

Table 2.1: Comparisons with Microprocessor Platforms

Platform	Algorithm	Clock(MHz)	Delay(us)	Energy(nJ/bit)
FPGA	RFSR	50	2.08	0.32
ATmega103	RC4	4	3262	105.12
StrongARM	RC5	206	53	41.41

The results and comparisons are shown in Table 2.1. The average energy consumptions of ATmega103 and StrongARM are 329 and 130 times more than that of the proposed RFSR scheme, respectively. Even though StrongARM is running 4 times faster, the delay is still 25 times larger than that of RFSR FPGA implementation while the delay of ATmega103 is 1568 times larger with 12.5 times slower clock than those of the proposed.

Comparing with FPGA, ASIC implementation runs faster and is more energy efficient but much more expensive to prototype. Existing research [48] compares FPGA and ASIC designs in circuit speed and power consumption and shows that ASIC designs are 87 and 14 times better than FPGA design, in static and dynamic power consumption, respectively. The proposed scheme therefore uses even less power with ASIC implementation. Even though the comparisons in Table 2.1 are based on different platforms and different encryption algorithms, it is clear that the hardware-oriented RFSR scheme is better suited for use in sensor nodes due to low energy consumption and small delay.

Table 2.2: Cipher Comparisons on FPGA

Algorithm	Logic Elements	Delay (us)	Energy (nJ/bit)
RFSR	5207	2.08	0.56
RC4	12917	6.40	11.18
RC5	6172	18.56	7.75

We also implement RC4, RC5 and the proposed RFSR cipher on the same FPGA platform to make the comparisons fair. The tests are run on Altera Cyclone II EP2C15AF256A7 at 50 MHz clock rate. According to the results shown in Table 2.2, the hardware-oriented RFSR cipher entirely outperforms RC4 and RC5.

2.4.3 Comparison with Grain 128

The proposed design is compared with Grain 128 on the energy consumption of keystream generation with 10 MHz clock and 8 bit/clock throughput rate. The results are comparable: 0.253 nJ/bit for Grain 128 and 0.544 nJ/bit for the RFSR scheme. However, considering the transmission overheads caused by the IV size, the energy consumption of the proposed RFSR scheme is 10.3% lower than Grain 128 for a packet size of 512 bits. Besides, to break RFSR by brute force, it requires about $1.33 * 2^{34}$ times more complexity than for Grain 128.

2.5 Conclusion

In this chapter, we have proposed a low complexity reconfigurable feedback shift register based stream cipher RFSR and shown that it is more secure than the widely used Grain, RC4 and RC5 algorithms. Implemented on an FPGA platform, the proposed scheme consumes over 130 times less average energy, and renders over 25 times less delay than existing microprocessor platforms.

Chapter 3

RFSR Cipher Based Authentication Protocol and Link Layer Encryption

In this chapter, an RFSR cipher based authentication protocol (RAuth) and an RFSR cipher based secure sensor network communication architecture (RSec) are proposed.

A WSN is special compared to a traditional computer network. The many constraints inherent in WSN often make it inefficient and sometimes impossible to use the existing network security mechanisms directly. Hence, there is an urge of developing security approaches specifically for a sensor network.

WSNs were first designed to perform military tasks such as battlefield surveillance, monitoring and sensing. Later, such networks were used widely in industrial and consumer applications, such as industrial process monitoring and control, and so on. In recent years, a new trend of smart home and smart office brings WSNs into our living and working spaces by making functional home and office appliances smart and accessible. In order to achieve this, a mechanism is in need to connect all the appliances together so that they can send and receive messages securely. A WSN is born for this purpose since it is not only reliable but also small and easy enough to be integrated.

As analyzed in the previous chapter, the RFSR cipher has low power consumption and high encryption speed which makes it a suitable cryptography algorithm in WSNs. In this chapter, an RFSR cipher based link layer secure communication architecture is proposed. Unlike the traditional ciphers, since RFSR uses a dynamic feedback structure, the encryption algorithm is considered to be a secret as the key. This unique property enables an RFSR cipher to get rid of the use of initial vectors (IVs) to reduce communication overheads. Without IVs, RSec employs a new mechanism to synchronize the sender and receiver with much lower communication overheads.

RSec helps two nodes to safely communicate with each other after they verify each other's identity. The RFSR based authentication protocol RAuth is used to help the two nodes to establish trust in each other. Since different RFSR ciphers have different cipher structures and secret keys, a trust center is needed to help two RFSR ciphers to authenticate each other and assign a new cipher to both nodes as the pairwise encryption credentials.

The remainder of the chapter is organized as follows. Section 3.1 introduces the network topology considered in this chapter. Section 3.2 presents the RFSR cipher based authentication protocol. The RSec link layer encryption architecture is proposed in Section 3.3. The performance evaluation is discussed in Section 3.4. Finally, Section 3.5 concludes this chapter.

3.1 Network Topology

Based on the topology, a wireless network can be centralized or distributed. Centralized networks, such as shown in Fig. 3.1, are common in our daily lives, such as Wi-Fi and the cellular system. Usually, a Wi-Fi network has one or several access points (APs) to which all the devices in a network connect. APs act as a central transmitter and receiver of wireless radio signals. All the devices directly connect to the APs and communicate only with the AP. Similar to Wi-Fi networks, the APs in the cellular networks are base transceiver stations (BTSs). All the user equipment (UEs) connects to the BTSs to get access to the network.

Distributed networks, as shown in Fig. 3.2 are widely proposed and discussed in WSNs. Without a base station, nodes in the network share the routing information with each other and establish connections using certain routing protocols. Distributed networks are more suitable to be deployed in hostile environments. In such situations, a centralized network can be easily destroyed by attacking the central node. After deployment, sensor nodes start the authentication phase to establish secure connections with their neighbor nodes by verifying if common secure credentials are shared.

Distributed networks are easy to use. After deployment, nodes will automatically authenticate with each other and find the routes on their own. The side-effect of this

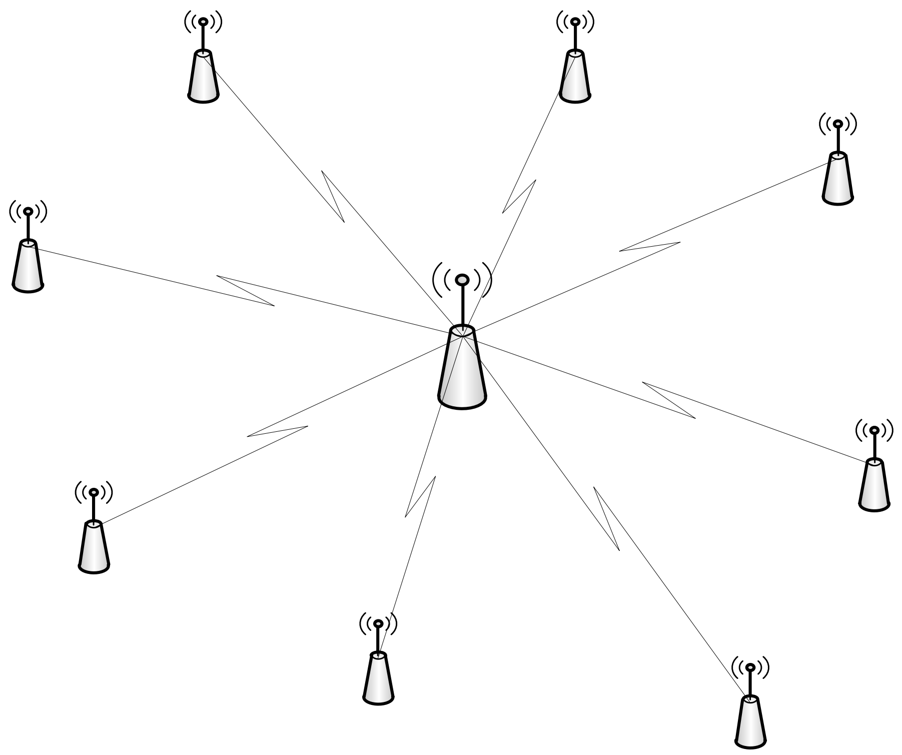

Figure 3.1: WSN centralized topology

high automation is that the overhead is large. One node first needs to authenticate with its neighboring nodes to establish secure links. After that, the routing tables are shared among the nodes which becomes a heavy task for energy constrained WSNs, especially when the scale grows. On the other hand, the dynamic routing protocols are robust against single node failure. When a node stops working, the traffic can be routed in another circuit to the same destination automatically. Besides, the energy consumption is relatively even. Since the routing protocol is highly flexible, a node can choose to reserve energy by minimizing the routing tasks when the energy drops.

On the other hand, a centralized network has more instinctive authentication and routing protocols. Routing is simple since the traffic either starts from or ends on the base station. Each node only needs to show its credential to the base station, and then the authentication is done. The centralized network, such as Wi-Fi, usually suffers from throughput degradation when the number of clients increases [49]. Besides, the centralized networks are prone to attacks when the target is the base station. The

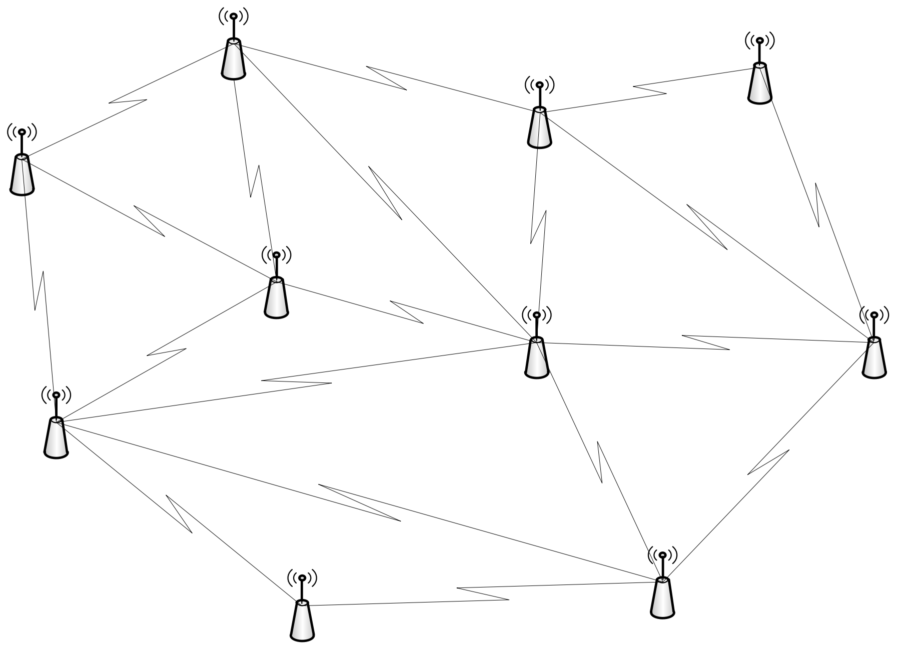

Figure 3.2: WSN distributed topology

energy consumption in a centralized network is uneven. If the base station has the same energy constraint as a regular sensor node, it will soon be exhausted.

Taking the application context into consideration, some of the assumptions will be different from the ones discussed above. Considering deploying a WSN in home or office environment, some of the sensor node can have constant power supply so that the power constraint does not apply to these nodes anymore. The nodes with only the internal battery can use these nodes as a relay to save energy in the case that the base station can only be reached with high transit power. Home and office environments have a lot of human activities which makes physical attacks targeting the base station less effective.

A WSN with centralized topology seems to be a good choice to this application context. But why not use the existing Wi-Fi to form a WSN? The reasons are stated as follows. Firstly, when the total number of nodes grows, Wi-Fi suffers from severe performance degradation which will affect its major functionality. Secondly, sensor nodes have limited transmission power. It's quite possible that a number of sensor nodes cannot send packets directly to APs. Thirdly, Wi-Fi requires time synchronization which is a power consuming task for sensor nodes. The Wi-Fi radio module is more expensive in the power perspective, and the transmission range of 2.4 GHz Wi-Fi is comparably smaller than that of the 915 MHz ISM band with the same TX power due to the larger path loss for a higher frequency. From the empirical tests results [50], half of the operating frequency can provide a doubled range.

The proposed hierarchical centralized topology is illustrated in Fig. 3.3. The base station (BS) is a powerful sensor node that has sufficient computational ability and external energy supply. A sink node (sink) is a sensor node with less powerful ability to reduce the cost but with external power supply. The sensor nodes are the ones that have limited energy and perform simple tasks. Based on the topology scenario, if there are sufficient sinks in the network, some of them can perform as a regular sensor node. Not only the sinks, sensor nodes can also connect directly with BS.

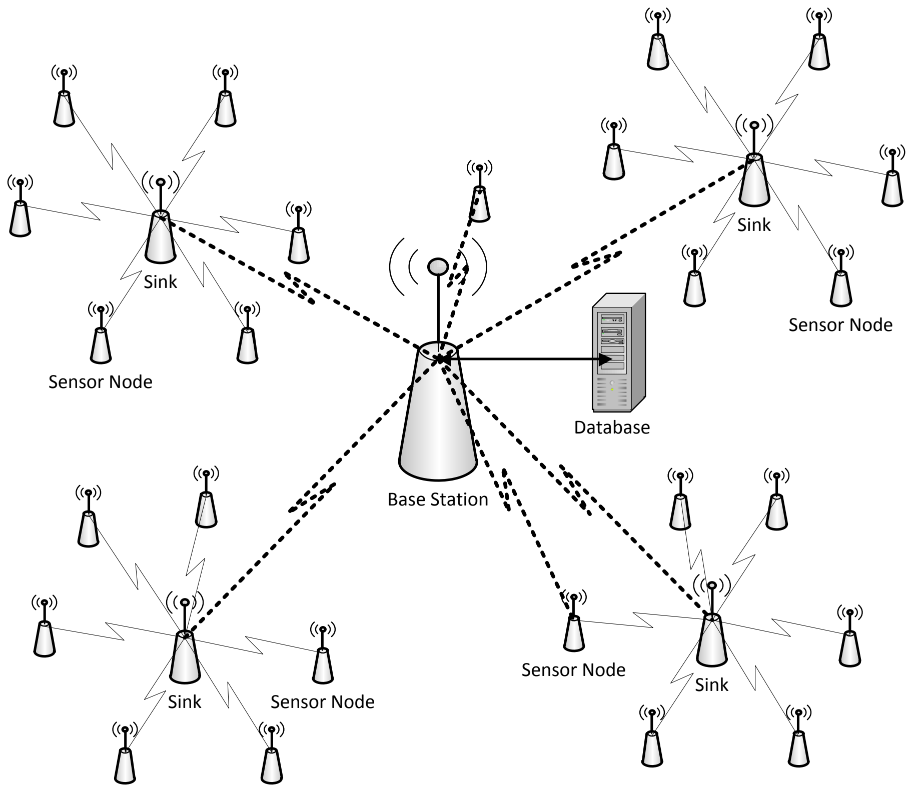

Figure 3.3: RFSR topology

Compared with the distributed network, the proposed topology avoids the necessity of sharing routing tables among sensor nodes. Nodes only communicate with the base station directly or indirectly with a simple route table. When the scale becomes larger, the hierarchical structure provides extended benefits to network formation.

Compared with the centralized network, the proposed topology has a larger coverage with the help of the sink's relaying functionality. The sensor nodes save much transmission power to communicate with the sink nearby, rather than the base station far away.

The proposed topology is similar to the cluster tree topology of Zigbee but a slight difference exists. In Zigbee, the sink can use another sink to relay messages, while in the proposed topology, a sink should directly connect to the base station. Besides the cluster tree topology, Zigbee also supports a star and peer-to-peer (mesh) network topology for different user contexts while the proposed authentication protocol and link layer architecture only supports one topology to simplify design complexity.

3.2 Authentication Protocol RAuth

Authentication is essential to both wireless sensor networks and sensor nodes. For a wireless sensor network, authentication guarantees that only the legal nodes are permitted to join in the network, which means all the nodes within the network are valid and can be trusted by other nodes. For a sensor node, authentication verifies the identity of the existing network so that the node can trust the other nodes within the network. Therefore, authentication is a bridge of trust which connects a sensor node and a sensor network.

An authentication protocol is a mechanism that helps nodes to establish trust with each other and helps nodes to form a sensor network. Nodes will exchange certification material and form connections with each other. The certifications are commonly preloaded into each node or entered during node deployment. It should be unique so that two nodes will never have identical certifications. Also, it should be sufficiently secure and extremely hard or technically impossible to crack. Besides, it should be recognizable only to legal nodes while no useful information is revealed to irrelevant nodes.

An RAuth protocol takes advantage of RFSR cipher's dynamic property to carry out node authentication. Each node has a unique ID different from any other node's within a network. A specific feedback structure and the initial key are applied to each node in the manufacture phase. Therefore, each sensor node has its specific ID, cipher feedback structure and initial key. The information is also saved remotely in the database. Whenever authentication starts, nodes will be verified by checking the ID and cipher information provided by the joining node with the information saved in the database.

To be specific, a unique 64-bit node ID is pre-loaded into each sensor node during the manufacture process. The RFSR cipher structure information and corresponding initial key are also loaded as part of the security certification. These are considered as the physical identification of a sensor node which cannot be modified afterwards. In order to safely authenticate itself to other nodes, the secure identification information is not supposed to be revealed. However, based on the secret RFSR cipher structure and initial key, a node can produce a bit stream to help other nodes identify its identity. In short, in an authentication process, a bit stream generated from the key and structure is the certification material to verify if the node is what it claims to be.

Only BS has access to the database which stores the nodes' credential information. Sinks will help those nodes who have limited send power, or limited receive resolution, by relaying packets between BS and nodes. A node can connect directly to the BS or use a sink as a relay.

3.2.1 Nodes' Credentials

In order to join the network, a node needs to prove that it is legal. The authentication is in two directions. A node needs to convince the base station with its credentials, and it also needs the base station to provide the credentials to verify a genuine base station.

The credentials in use are the cipher initial feedback structure and the pre-loaded secret key. When the authentication process starts, a node will generate a random bit stream with the cipher structure and initial key. The random bit stream together with the node's ID, will be transmitted to the base station. The base station will checkout the nodes initial feedback structure and key by searching the database. Afterwards, the base station examines the node's identity by comparing the random bit stream received with the bit stream it generates after loading the corresponding initial information. If the two bit streams are identical, the base station can confirm that the node is who it claims to be. If not, the authentication process is terminated.

The next step is that the base station will provide authentication information to the node. Similarly, the base station will continue to produce a random bit stream with the same cipher structure and send it to the node. The node receives the bit stream and checks if it is valid in the same way the base station generates it. If the verification succeeds, the node will admit the identity of the base station.

Several variations may apply to the authentication process:

First, besides the base station and the sensor node, additional nodes may participate in the process acting as relaying nodes to help the base station and the sensor node establish a physical link circuit. The link can even be asymmetric [51], which means the uplink, from the node with lower hierarchy to the node with higher hierarchy, and the downlink, from the high hierarchy node to the low hierarchy node, can be different. Regardless of the actual situation, the link is transparent to the authentication process.

Second, more than one cipher structure and initial key pair for authentication can be stored on a node. For certain situations where the environment is hostile or the authentication robustness is essential, a backup credential pair is a good design. Whenever the first cipher pair is considered used up, the second cipher pair will be used as default.

Third, the procedure of generating the random bit stream for authentication can vary according to the requirements. In order to resist replay attacks, the node needs to have the ability to send different random bit streams for authentication based on the received challenge bit stream. The challenge bit stream, usually called the nonce, is a one time random bit stream for cryptographic communication. If the node is given a nonce to use in the authentication bit stream generation process, the generated random bit stream is specific for each given nonce. The adversary has the ability to get the authentication bit stream and the corresponding nonce by eavesdropping on the communications. However, even with this information, it still cannot fake the authentication bit stream for another nonce. Also, there is no effective way to recover the key and structure based on the eavesdropping information.

In the RAuth protocol, the nonce is a 32-bit string. The nonce is first XORed with the first 32-bit of the initial key. Concatenated with the last 64-bit initial key, the 96-bit string is then loaded as the initial internal state of the RFSR cipher. After cipher initialization, the next 32-bit will be used as the authentication credential.

3.2.2 New Node Joining an RAuth Network

The detailed procedure of how RAuth protocol works when a new node joins an existing WSN is described below.

The authentication can be either active or passive as per the configuration of the base station. In an active authentication network shown in Fig. 3.4, the base station

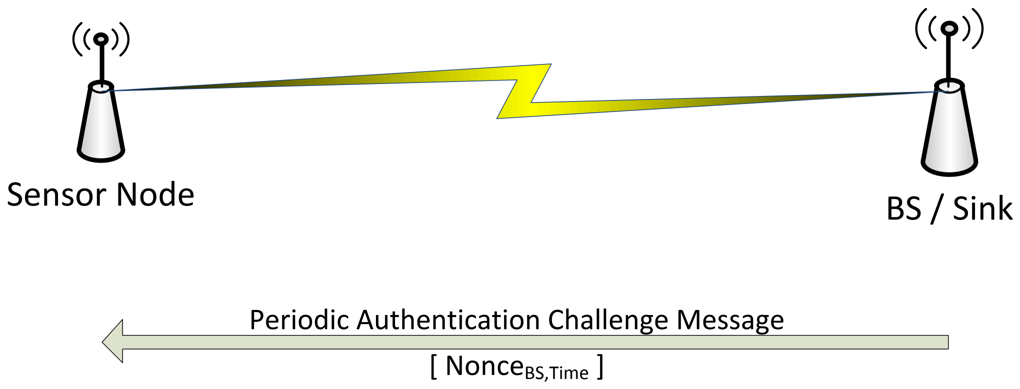

Figure 3.4: RAuth active mode

and the sinks will periodically broadcast an authentication challenge message. In this message, a nonce is included, together with the radio module information such as transmission power, minimum receivable power and so on. When a sensor node receives this broadcast message, the path loss can be roughly calculated based on the received signal strength indicator (RSSI) from the node's radio module and the sender transmission power in the received message. Therefore, it is not hard to expect whether the sensor node has enough transmission power to directly deliver the authentication message based on the roughly calculated path loss, the node's transmit power and the base station's minimum receivable power. $Nonce$ is the random bit stream which can be pseudo-randomly generated by a shift register structure inside the broadcasting node, or genuinely randomly generated by using time information, or the channel status captured by radio, etc.

In a passive authentication network shown in Fig. 3.5, the authentication process is initialized by node N broadcasting a new node authentication request. By receiving the request, the base station and the sinks will unicast to node N the same message which they broadcast in the active mode.

Since the active and passive mode have different gross overheads in different periods of time, the choice of an active or passive mode can vary in different phases of a network formation. When the network is just initialized and nodes are beginning to authenticate, it is more efficient to use an active mode because it is likely that more nodes than one are listening to the traffic for the broadcast challenge messages, and one broadcast challenge message from the base station or a sink can provide

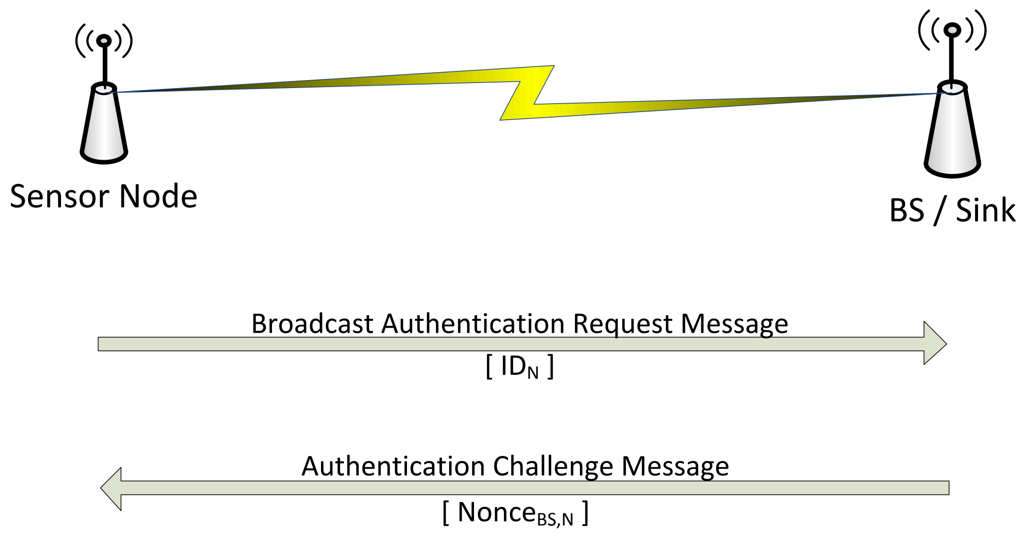
Sensor Node
BS / Sink
Broadcast Authentication Request Message
[ID_N]
Authentication Challenge Message
[$Nonce_{BS,N}$]

Figure 3.5: RAuth passive mode

several nodes with the information of available parent nodes. When a large number of nodes are already authenticated, the passive mode can be more efficient. From the channel usage perspective, the deletion of broadcasting will give the time slot to the transmission of regular packets which increases the gross network throughput and therefore makes the network initialization faster. From the energy perspective, the passive mode not only saves the transmission energy of the base station and the sinks, it also saves the energy for all the sensor nodes within the coverage of the broadcast signal, since receiving can be as energy consuming as transmitting.

When the new node N is deployed in the environment, in the active mode it will start to listen to the traffic for the broadcast authentication messages. When N receives the message, it can also obtain the receive signal strength indicator (RSSI) which is supposed to be supported by the radio module of the sensor node. Assume that the radio environment is symmetric, which means that receiving and transmitting channels are identical, N is capable of calculating if the BS can receive the packet it sends. If so, a direct link can be established. If not, N needs a relaying node for the uplink. In the passive mode, N will broadcast a request and wait for the response. The base station and the sinks who receive N's request will respond directly to N with the challenge message nonce.

The connection status between the new node N and the base station BS can be classified into three categories. The first situation is that a direct connection exists between N and BS, which means that BS can receive the packets sent from N and N can also receive the packets sent from BS. The second situation is that N and BS need a third node to help relay packets in between. The last situation is that N can receive packets from BS but BS cannot receive packets from N. This is a normal case when different hardware is used on base stations and regular sensor nodes so that the transmit power and minimum receivable power are different. Therefore, the uplink requires a third node to send packets from N to BS while the downlink is direct from BS to N.

Direct Link

When N has enough transmit power to communicate directly with the BS, the direct link method is efficient and straightforward. The active mode and passive mode only differ in the first step of the RAuth protocol. In either way, N will receive a 32-bit random bit stream $Nonce_{BS,N}$ from the BS. The detailed authentication message flow is shown in Fig. 3.6. For the RFSR cipher with 32-bit LFSR and 64-bit NFSR, $Nonce_{BS,N}$ will XOR with LFSR's initial key. The result is loaded in LFSR as the new initial state. After 96 clocks running as cipher initialization, $Nonce_{BS}$ bits will influence all the internal 96 bits. Then, the next 64 clocks will produce a 64-bit authentication response $RESP_{N,BS}$. The actual message from N to BS will be $[ID_N, RESP_{N,BS}]$.

In the same way, N should also give BS a challenge nonce so that only the genuine BS can come up with the correct response. Due to the fact that BS and N share the same cipher structure and initial key, $Nonce_{N,BS}$ can use the 32-bit stream known to both parties. Therefore, one choice is that N's RFSR cipher will continue to produce 32-bit stream as $Nonce_{N,BS}$. N will store this $Nonce_{N,BS}$ in temporary memory to wait for the response from the BS.

When the BS receives the response from N, it will look in the database with ID_N for the corresponding RFSR cipher structure. The BS produces a 64-bit stream by following the same routine N uses to produce $RESP_{N,BS}$ and checks if the two streams match. If not, the authentication is aborted. If so, the same as N, BS will continue to produce 32 bits used as $Nonce_{N,BS}$. It will load N's initial key again, XOR the 32-bit in LFSR and then repeat the previous procedure to produce response $RESP_{BS,N}$. The reply message is $[RESP_{BS,N}]$. Once N receives and verifies this

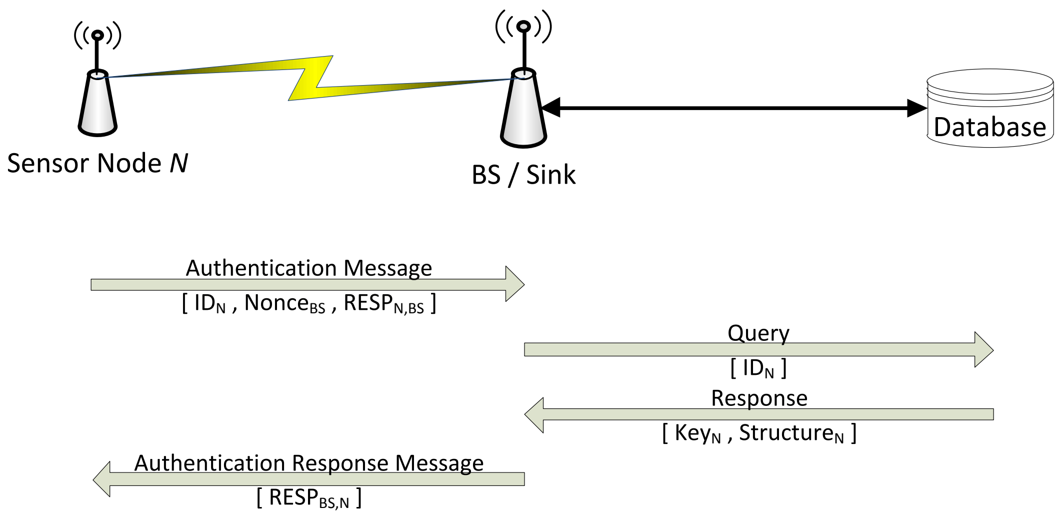
Sensor Node *N*
BS / Sink
Database
Authentication Message
[ID$_N$, Nonce$_{BS}$, RESP$_{N,BS}$]
Query
[ID$_N$]
Response
[Key$_N$, Structure$_N$]
Authentication Response Message
[RESP$_{BS,N}$]

Figure 3.6: RAuth direct link mode

message, the authentication process is over.

Indirect Link

It is a common situation that the coverage of a base station is limited and needs other nodes to act as tiny base stations to help increase coverage. A sink is a sensor node that functions like a base station but is less powerful and can serve a limited number of sensor nodes. In a situation that a base station and a sensor node cannot connect directly, a sink in between relays messages so that they can connect indirectly.

In the case of an indirect link, passive and active modes only effect the first step. In the active mode, N listens to the broadcast challenge messages and connects to the best sink based on certain criteria. In the passive mode, N broadcasts its authentication request and chooses one sink from the replying sinks on the same criteria. Assume that sink S is chosen as the parent of N. The challenge that N receives is $Nonce_{S,N}$. The shared RFSR cipher between S and BS is $C_{BS,S}$. The detailed authentication message flow is shown in Fig. 3.7.

N will reply with the response $RESP_{N,S}$ generated with its authentication RFSR cipher, initial key and $Nonce_{S,N}$. Since S has no information about N's authentication credentials, it cannot verify the identity of node N. The challenge $Nonce_{S,N}$ and the response $RESP_{N,S}$ will be forwarded to BS in the form of $[Nonce_{S,N}, ID_N, RESP_{N,S}]$.

BS will verify the credentials the same way as a direct link. The reply from

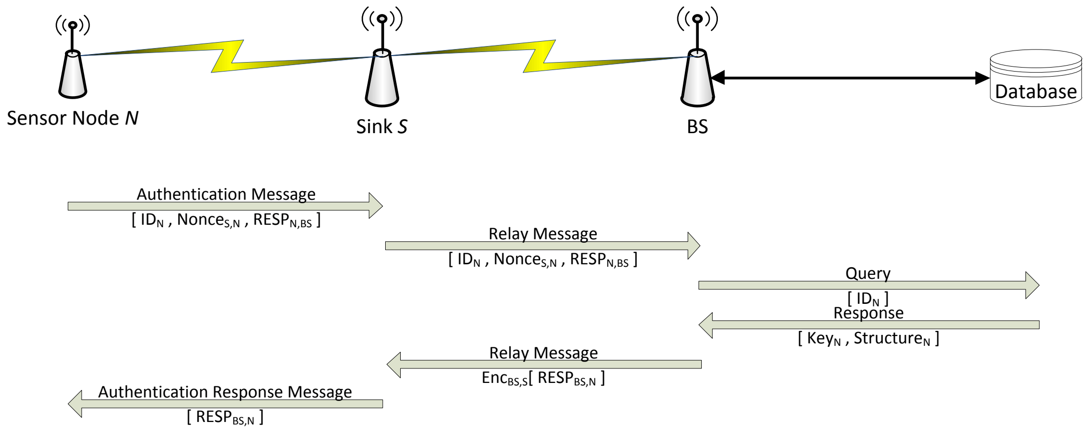

Figure 3.7: RAuth indirect link mode

BS to S is $Enc_{BS,S}([RESP_{BS,N}])$. S decrypts the message and sends the response $RESP_{BS,N}$ to N. Once N receives and validates the message, the authentication process has succeeded.

Asymmetric Link

Except for the cases of direct and indirect links, it is common that a sensor node can receive messages from a base station since a base station usually has powerful hardware and larger transmission power. On the other hand, the sensor nodes are usually power constrained devices which have limited transmit power to reach the base station. Normally, if there is no direct link, an indirect link is utilized by using a third node as relay node for both directions. It obviously incurs unnecessary overhead because BS is capable of sending a message directly to N but instead of that, a relay node is used to receive and forward messages in the middle. The overhead exists in both channel usage and power consumption. In order to optimize this situation, an asymmetric link is used, which means that the uplink and the downlink are not identical anymore.

How would a node know when to use an asymmetric link? In the active mode, N can receive BS's broadcast messages with BS' transmit power parameter. By comparing with the RSSI and transmit power of N itself, it can know if direct link or asymmetric link should be used. In the passive mode, N will also listen to the channel first. If it senses messages sent from the BS, i.e., a node with ID 1, it can confirm that it is in BS's coverage area. N will then broadcast the authentication request. From all the reply messages, if N receives a reply from BS, a direct link will be used. If not, it becomes clear that an asymmetric link should be used.

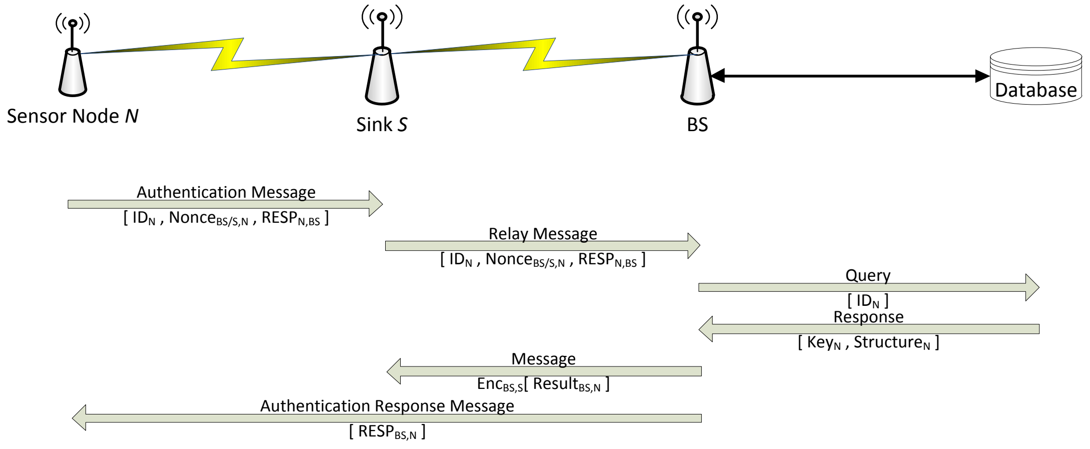

Figure 3.8: RAuth asymmetric link mode

In either mode, an authentication challenge $Nonce_{BS/S,N}$ is received from BS or a sink S. If it is received from BS, then N also needs to pick a sink S as its parent to help relay packets. The authentication procedure is similar as displayed in Fig. 3.8. The challenge response will be $RESP_{N,BS/S}$ which is sent to S. If in the active mode, S will keep a copy of $Nonce_{BS}$ in local memory. If the received nonce is valid, S will encrypt and forward the message $Enc_{BS,S}([Nonce_{BS/S,N}, ID_N, RESP_{N,BS/S}])$ to BS.

BS will verify the response. If correct, BS will send a message directly to N with $RESP_{BS,N}$. In parallel, BS will send a message to S to notify the result of authentication. Once N verifies the response from BS, the authentication process is finished.

3.2.3 RFSR Cipher Management

To protect the communication confidentiality in the network, RFSR ciphers are used to encrypt data before they are sent. It is required that the sender and the receiver share the same cipher information, including the RFSR cipher feedback structure and the cipher internal state. Using the same RFSR ciphers guarantees that the sender and the receiver can produce the same keystream so that the message before encryption at the sender side is identical to the message after decryption at the receiver side.

In the hardware architecture, each sensor node is equipped with one RFSR cipher hardware and stores all the cipher information it shares with its link neighbor. When a new packet is received or a new packet is to be sent, the node will load the required cipher information to the hardware for the purpose of decryption or encryption.

Cipher Information in Storage for a Link

The cipher information consists of two parts. One is the cipher feedback structure. The other is the cipher internal state. Based on the topology hierarchy, the communication direction between two nodes are defined as the uplink and the downlink. For both directions, the cipher feedback structure is the same while the internal states are different. For each cipher on each node, two sets of cipher information are managed separately for uplink and downlink. After each use of cipher, the cipher information will be saved to handle packet loss situations, which will be explained further in the next section.

Each cipher maintains a cipher with the base station, which is called the master cipher. The master cipher is pre-loaded in the manufacture phase and is unique for each sensor node. The cipher information management requires storage space to save the cipher information. For example, the sensor node N connects to the base station BS indirectly with the help of a relaying sink node S. For N, a master cipher with BS and a link cipher with S are stored in N's memory. To send a message to BS, N first encrypts the message with the master cipher. After that, N will save the current master cipher status and load the link cipher from the memory. The link cipher is required to encrypt the message to make sure that S can verify that the message is from a legal node.

Once the authentication is finished, the new cipher information will be assigned and used to encrypt messages later on. As mentioned above, a counter instead of IV is transmitted in all the link layer packets. The counter is tightly related to the RFSR cipher status.

The maximum length of the link layer packet in an RSec system is fixed. To introduce flexibility, the maximum length can vary based on the type of sensor nodes in the link and the signal strength. For communications between two sensor nodes with sufficient transmit power and in the right range, the packet length can be larger.

Cipher Information Assignment

After the RAuth protocol helps two sensor nodes authenticate each other, a new cipher will be assigned by the base station for later packet encryption use. The new cipher information includes the cipher key and the cipher structure. Besides, a 32-bit IV will be generated by one of the two nodes with higher hierarchy. The IV will be sent unencrypted to the node with lower hierarchy. Then the two nodes will have exactly the same cipher IV, key and structure. Based on this information, two nodes will initialize the new cipher information by generating initial internal states for uplink and downlink and save them locally. In order to know whether the other node has finished the cipher information initialization process, handshake messages are sent which are encrypted with the new cipher information. Depending on the topology of the two nodes, the handshake message can be an acknoledgement (ACK) packet, an ACK packet responding to another ACK packet, or a simple handshake request packet. Successful verification on these messages will notify both nodes that the other node has already used the new cipher and the message should be encrypted with the cipher later on.

Since the system supports direct link, indirect link and asymmetric link, the new cipher assigning process differs for each node connection topology.

For the direct link topology, the new cipher information is appended to the reply message from BS to N. The reply message will become

$$[RESP_{BS,N}, IV, Enc_{BS,N}(Key, Structure)].$$

When N receives the reply message and validates the content, it will load the new cipher and send an encrypted ACK message with the new cipher.

For the indirect link topology, the new cipher information is also appended to the reply message from BS. BS needs to give the cipher information to both the sink S and the node N. The cipher information for node N will be encrypted using N's RAuth cipher. The reply message from BS to S will be

$$Enc_{BS,S}([RESP_{BS,N}, Key, Structure, Enc_{BS,N}([Key, Structure])]).$$

Once S receives this message, it will append IV and forward the message to N. Finally, N will receive $[RESP_{BS,N}, IV, Enc_{BS,N}([Key, Structure])]$.

For the asymmetric link topology, the new cipher information is sent separately to the sink S and the node N. N will receive the BS message

$$[RESP_{BS,N}, Enc_{BS,N}([Key, Structure])]$$

while S will receive the BS message $Enc_{BS,S}([ID_N, Key, Structure])$. S will use the new cipher information to send a handshake message to N to make sure the new cipher can be used, which is $[IV, Enc_{S,N}(Handshake)]$. N loads the IV and verifies the handshake message. If it is legitimate, an encrypted ACK will be responded. The cipher assignment process finishes.

Cipher Information Update

A key update is an effective way for stream ciphers to avoid reusing IV, which will jeopardize the security of the communication system. For the systems, even though the key-IV is not used the same way as in a common stream cipher system, a cipher information update is also effective to prevent brute-force and other attacks. Except for the secret key, which is used by a traditional stream cipher, the dynamic feedback structure adds a new dimension to the secrecy of the cipher. The RFSR cipher has the ability to update partial or total credentials. Considering to update the cipher structure, since RFSR is composed of an LFSR and an NFSR, updating any one of them will result in a totally new cipher. Therefore, the feedback function of LFSR and NFSR can also be updated separately. To conclude, instead of a key update, the cipher update can be total or partial; partial updates can choose to update one or any combination of two from the LFSR feedback function, the RFSR feedback function and the secret key.

A flexible cipher update mechanism is provided, which fits a variety of environments. The cipher information update procedures are similar to the cipher information assignment. Instead of sending authentication messages, both nodes will notify the base station that a cipher information update is requested. The base station will send the cipher update message to both nodes in the similar way described above for all three topologies.

The RFSR cipher credential information consists of a secret key and three reconfigurable functions, i.e. a linear feedback function, a non-linear feedback function and an output function. Updating of one of these four pieces of information will produce a complete new cipher credential. Therefore, the update decision is flexible, being either a partial or a complete update. According to Section 2.2.4, one or several of a 64-bit key, a 25-bit linear feedback function information, a 119-bit non-linear feedback function information and a 73-bit output function information can be used to perform the cipher update task. The cipher information update process will bring transmission overheads. Compared with other key management protocols, which will periodically update the keys, partial cipher update of the RAuth protocol has the similar or smaller communications overhead.

3.3 RSec Link Layer Encryption

The link layer provides addressing and channel access control mechanisms that make it possible for all the sensor nodes within a network to communicate with each other. Since sensor nodes in a WSN have limited energy and computational ability, the OSI model or Internet Protocol Suite layered systems are much too heavy for it. The WSN described here is considered to be composed of three layers: the physical layer, the link layer and the application layer. The RAuth protocol is considered part of the application layer. The physical layer consists of physical links and means of transmitting raw bits, which is not discussed here. In this thesis, we focus on the encryption perspective of the link layer.

3.3.1 Addressing

Each sensor node in a RSec system has a 64-bit node ID. Similar to the Media Access Control (MAC) address of the Internet, each network interface has a unique serial number. The node ID is not used directly in an RSec system because the transmission overhead is unnecessary and avoidable. An RSec system uses an 8-bit address $ADDR$ as the identification in a network. The address of the base station BS_{ADDR} is hard coded as 255 (0b1111 1111) in all RSec networks. The unauthenticated node address is 128 (0b1000 0000) which is used when a new node is added to a network and the authentication is not yet finished. It is possible that more than one sensor node are not authenticated in an RSec network at the same time. If that is the case, the node needs to set the first 64-bit of the packet payload as its node ID. If both the source node and the destination node are not authenticated, the destination node ID is set as the first 64-bit of the payload and the source node ID as the second 64-bit. To support in-network broadcast, 127 (0b0111 1111) is reserved for this purpose.

To minimize the chance of packet collision when nodes access the medium, RSec uses the carrier sense multiple access with collision avoidance (CSMA/CA) technique, which is utilized in IEEE 802.15/Zigbee and IEEE 802.11/Wi-Fi. Before each packet is transmitted, the sender will broadcast a request to send message (RTS). When the receiver gets this RTS message, it will broadcast a clear to send message (CTS). This mechanism will reduce the chances of packet collision, especially for the hidden node situations.

3.3.2 Packet Format

The packet format of an RSec system is shown in Fig. 3.9. The packet header consists of destination address, source address, message type, counter and packet length. The message type informs the receiver of the appropriate way to handle the payload of this packet. Only the header is not encrypted.

The RSec packet format A and B have the same header size and the message integrity code (MIC) size but different payload sizes. In the packet header, format A uses a 4-bit counter field and a 6-bit length field while format B uses a 3-bit counter field and a 7-bit length field. Multiple packet formats make the RSec system suitable in different applications.

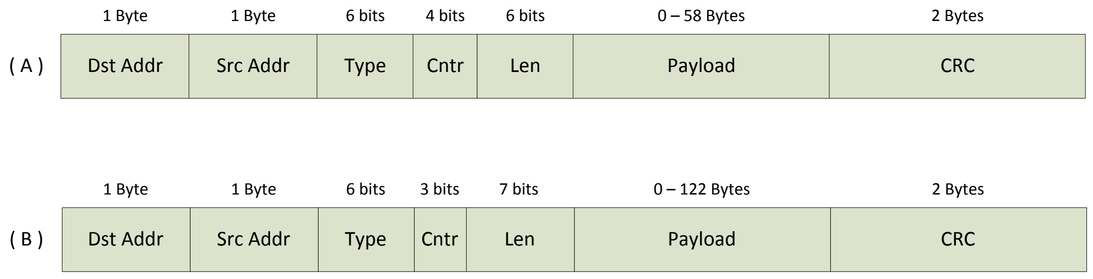

Figure 3.9: RSec packet format A and B

To detect transmission errors, MIC is used to add redundancy. A cyclic redundancy check (CRC) [52] algorithm is utilized as the message integration check algorithm. CRC is an error-detecting code used to detect accidental changes to raw data. Blocks of data entering the system get a short check value attached, based on the remainder of a polynomial division of the content; on retrieval the calculation is repeated, and corrective action can be taken against presumed data corruption if the check values do not match.

A message authentication code (MAC) is used to verity whether the source node of the packet is what it claims in the source address field. In an RSec system, the MAC is the MIC encrypted by the keystream.

Packet payload and the MIC are encrypted. The receiving node will first generate the keystream based on the information in the header and decrypt the message. Then, the received MAC is calculated by concatenating the packet header with the payload and applying the CRC algorithm to the concatenated string. By comparing the MIC received from the packet with the calculated MIC, the receiver will tell if the message is valid.

In conventional systems, only the payload of a packet is encrypted while the payload and MAC of a packet are both encrypted in RSec systems. This only works with stream ciphers. Since a stream cipher works by bit-wise XORing the original bit stream with the keystream, any bit flip over transmission will finally impact the decrypted message at the same place. A bit flip is the same as xoring the bit with 1. As shown in Fig. 3.10, assuming that 1011 is a 4-bit string part of the original message to be encrypted, 1100 is a 4-bit string part of the keystream which is to be XORed with the 4-bit string 1011 from the original message. The 4-bit string in the same place of the encrypted message will be 0111. Unfortunately, during the transmission, the third bit of the 4-bit string is flipped due to an interference and the receiver gets 0101 instead of 0111. The receiver node will use exactly the same keystream to decrypted the received message. The decrypted message then becomes $0101 xor 1100 = 1001$. If the same message is transmitted without encryption, after disruption at the same bit, the received 4-bit string will be 1001, which is exactly the same as the receiving string when the message is encrypted. This means that even with the encryption, the transmission error is still at the same place.

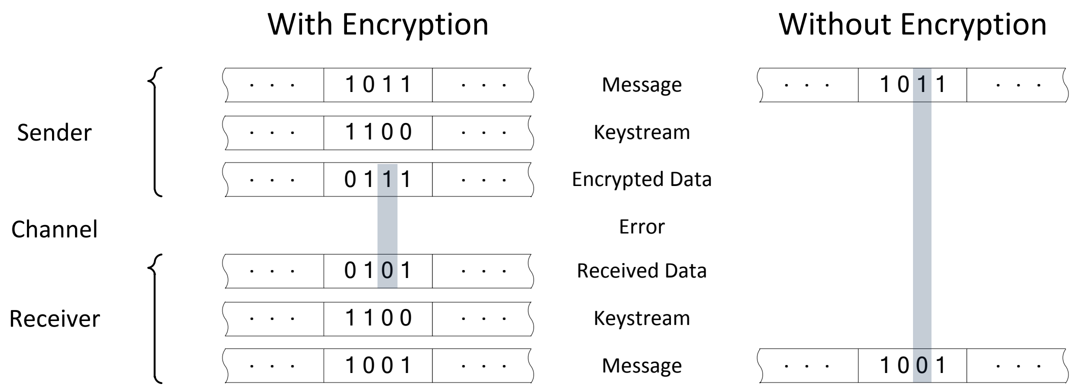

Figure 3.10: Bit flipping during transmission

The IV is used in a stream cipher as part of the initial states. The sender node concatenates each IV with the secret key and loads the new bit string into the cipher. Since each IV is only used for one particular packet, it makes the transmission stateless and any packet loss only affects the current packet but has no further influence, if we only consider it in the link layer. This design is suitable for a complex layered network, such as the Internet Protocol stack. Whenever a packet is lost in the link layer, its upper layer, which is in charge of the data completeness check, will send a retransmission request to notify the sender to retransmit the missing segment again. The retransmission overhead is large while the protocol can be clear and simple, which is an acceptable trade-off. In the aspect of a sensor network, the protocol is relatively less complex, and therefore such overhead is not hard to avoid.

In RSec systems, IV is not transmitted in clear text as a traditional stream cipher system. Instead, only a counter is sent. For RFSR systems, the source node sends a 4-bit counter rather than a 32-bit IV. It saves 87% of the IV transmission overhead. Two pair of counters are kept for both communication directions. If the uplink and downlink share the same cipher information, it is possible that one cipher state is reused by the two nodes in a communication link due to a packet loss. In this case, the keystreams are identical and, therefore, the contents of the messages are significantly jeopardized. To avoid this situation, the uplink and downlink should be logically separated. Even with the same cipher structure, the cipher initial internal states are different, so are the counters. Besides, a pair of counters for the base station are required to keep track of the master cipher status.

3.3.3 IV and Encryption

Traditional stream ciphers use IV as part of the cipher initial state. In this case, IVs are transmitted in clear text in each message. The advantage of this mechanism is that the receiver can verify the message in a simple way by loading the IV and the key as the cipher initial state. For a WSN, energy consumption is always the first consideration. To minimize transmission overhead, two new mechanisms without sending any IV are proposed in this thesis.

In either mode, counters are used to deal with packet loss and to synchronize the ciphers. For two nodes on a communication link, two counters are maintained for the uplink and the downlink, respectively.

Counter Index Mode

In this mode, the cipher works the same way as the conventional stream cipher. For each packet, a key-IV pair is loaded for encryption or decryption. Instead of transmitting the whole IV, only the counter, which is the index of the IV in use, is transmitted.

For each message, the cipher will first generate a keystream in the maximum length of the packet. Thereafter, the cipher will continue to generate 32 more bits than the IV for the next message. The cipher will save the current IV and counter in case of packet loss. Besides, the new generated IV will also be saved so that it can be used for the next time.

Similar to the sender, when the receiver gets a new packet, it will load the IV generated last time, together with the key, to generate the keystream to decrypt the received message. After that, a 32-bit keystream will be generated and saved for further use.

In all, when the cipher performs an encryption or decreption task, it generates not only the keystream used for the current packet, but also generates an IV for the next task. Each generated counter has an index between 0 to 7, assume that a 3-bit CNTR field is used. The sender and the receiver use the same algorithm to generate keystream and the IV. Therefore, only the index is sufficient for both parties to know which keystream should be used to perform an encryption or decryption task.

Continuous Mode

In continuous mode, IV is only used in the cipher initialization and the cipher resynchronization phase, which will be discussed later.

For regular encryption and decryption purposes, the cipher will generate a keystream in the maximum possible packet length. The 96-bit cipher internal states will be saved in the node's local memory. When the cipher is required the next time, the saved internal state will be loaded to the cipher and will be performed the same operation.

Considering packet loss situations, the new received counter may not be continuous with the last received one. In this case, the cipher needs to jump to the required status to perform the decryption task. For the continuous mode, the cipher will load the internal state and then continuously generate several maximum packet length keystreams. As for the counter index mode, the cipher should load the key-IV pair to the cipher, perform cipher initialization and then generate the keystream with the new IV. If the jump step is larger than 1, the cipher should load the new key-IV pair again and do the previously stated task for several times.

For the counter index mode, only a 32-bit IV should be temporarily saved; while in the continuous mode, a 96-bit internal state is required to be saved. For most of the cases, the 64-bit difference in memory consumption will not make a large difference. Besides, from experimental results, loading the 96-bit internal state, i.e., the 96-bit key-IV pair, is not negligible in neither energy nor delay perspective. Therefore, the continuous mode is chosen as the default mode to use counters in RSec systems, while the counter index mode is also supported.

3.3.4 Cipher Credentials and Encryption

As mentioned, each sensor node is pre-loaded with the cipher credential information, consisting of a secret key and a cipher feedback structure. This piece of cipher credential information is the master credential information and only used to perform authentication tasks with the base station. Apart from the master credential information, the session credential information will be generated when a node is authenticated, and used to encrypt or decrypt the communications thereafter. A session credential information is used between two direct connected nodes.

In case of a direct link connection, the sensor node and the base station first use the master cipher credential information to authentication each other and the new session credential information is generated. For later communcations, the session credential information is used to encrypt or decrypt the messages.

The indirect link connection is similar. The master cipher credential information shared between the sensor node and the base station guarantees the authentication. Then, two pieces of pairwise session credential information are generated, one is shared between the sensor node and the base station, the other is shared between the sensor node and the sink node which helps relay messages between the sensor node and the base station. Assume that the sensor node wants to send a message to the base station. The raw message will first be encrypted by its session credential information with the base station. The encrypted message will then be encrypted by its session credential information with the sink node. The double-encrypted message is forwarded to the sink node. When the sink node receives this message, it will decrypt it to verify whether it is from the valid sensor node. If so, the decrypted message will be encrypted again with the sink node's session credential information with the base station. When the base station receives the message, it first decrypts the message with the session credential information shared with the sink node and then with the one shared with the sensor node.

3.3.5 Cipher Information Initialization

Regardless of the topology, considering two nodes $N1$ and $N2$ in a communication link, the packet length is LEN, the key is $CKey$, the initial IV is CIV, the cipher structure in use is $CStruc_{N1,N2}$, the counters in use are $CCntr_{N1,UL}$, $CCntr_{N1,DL}$ and $CCntr_{N2,UL}$, $CCntr_{N2,DL}$, and the cipher states are $CState_{N1,UL}$, $CState_{N1,DL}$ and $CState_{N2,UL}$, $CState_{N2,DL}$.

For each cipher shared by two nodes, the uplink and downlink use the same cipher structure but different cipher internal states. As mentioned, the key and IV as the cipher initial state are shared by both nodes, which only provides for the uplink or the downlink. In RSec systems, the key and IV pair will be used as the initial state for either the uplink or the downlink, and a new cipher initial state for the other can be produced as follows. The cipher will load the key and IV, finish the initialization and use the next 96-bit as a mask to XOR with the 96-bit IV-Key pair. The result will be used as the initial state of the cipher for the other link direction.

If it is a direct link, assume that $N1$ is a base station and $N2$ is a sink or sensor node. $N1$ will first load CIV and $CKey$ and $CStruct$, run the initialization, output LEN bits keystream and use the first 96 bits as a mask. $CState_{N1,UL}$ is set with the result of the mask XORing with $CIV|CKey$. $CState_{N1,DL}$ is set with the current cipher internal state. In the final stage of authentication, $Ckey$, CIV and $CStruc_{N1,N2}$ are encrypted with the master key and sent from $N1$ to $N2$. Besides, all counters of $N1$ and $N2$ are initially set to 0.

After $N2$ receives and decrypts the message with the master cipher, it loads CIV, $CKey$ and $CStruct$ the same way $N1$ did and uses the same mechanism to generate $CState_{N2,UL}$. $N2$ saves the current internal state as $CState_{N2,DL}$ and loads the $CState_{N2,UL}$ as the new cipher internal state. After initialization, $N2$ generates LEN bits keystream. The first 32-bit are used as the payload of the ACK message. When $N1$ receives the ACK message, it loads $CState_{N1,UL}$, runs the initialization and verifies the ACK message. If the message is correct, $N1$ will know that $N2$ has already implemented the cipher $Cipher_{N1,N2}$.

To ensure $N2$ that the ACK is correctly received, $N1$ will send an ACK encrypted with the downlink cipher in counter 2. If the message is received by $N2$, the process is finished successfully.

For the cases of indirect links, the procedure is similar. Assume that $N1$ is a sink node and $N2$ is a sensor node. The difference is that $N1$ will receive the new cipher information from the BS rather than generate it by itself. This process is transparent to $N2$.

If it is the asymmetric link, assume that $N1$ is a sink node and $N2$ is a sensor node. The new cipher information is sent from BS to $N1$ and $N2$ separately. The two nodes will set the downlink and uplink states and counters based on the received messages. Then, a handshake is required. $N1$ will drive a handshake process by sending a handshake request. This handshake message will be encrypted with the new downlink cipher, and the counter is set to 1. $N2$ will load the downlink state and cipher structure to the hardware and decrypt the message. After the verification, the current cipher internal state will be saved as $CState_{N2,DL}$ and an ACK will be sent encrypted by the uplink cipher with counter 1. If the message is validated by $N1$, an ACK will be sent to notify $N2$ that the new cipher can be used to encrypt/decrypt communications later on.

3.3.6 Packet Loss Handling

Due to the uncertainty and asynchronous nature of sensor network radio communications, it is possible that a packet is lost due to collision or failure of checksum at the receiver, etc. In this case, cipher information management should be robust enough to handle this situation. In an RSec system, for each link on each node, two copies are saved in the memory. One keeps track of the sending status of the cipher and the other of the receiving status of the cipher. The sending and the receiving cipher share the same cipher feedback structure but have different cipher internal states.

Packet loss is commonly a major cause of performance degradation of low power wireless communication systems. Retransmission is required for most of the cases to make sure all sessions of an application layer message is successfully received by the end node. Therefore, efficiently handling packet loss and packet retransmission is considered as an important topic in the design of RSec systems. To ensure no packet loss on the link layer, an acknowledgment (ACK) mechanism can be utilized. If the sender does not receive the ACK from the receiver, it will assume that the packet is lost and retransmission will be required. On the other hand, if the data transmitted is tolerable to packet loss, an ACK will not be required which can save energy.

When the sender's link layer receives a transmission request from its application layer, based on the destination node address, it will load the corresponding cipher information from memory to the cipher hardware. The keystream is then generated and XORed with the raw message to create the encrypted payload field for the link layer packet. For the lossless mode, which guarentees no packet loss, a timer is used to pop an alert when the ACK is not received in a certain time. Once the timer expires and no ACK is received, the sending node will retransmit the packet. In the new packet, the counter field is increased from the counter in the previous packet by one. A new keystream is generated to encrypt the message.

In an RSec system, the 4-bit counter has 16 different values. The sender and the receiver keep a local copy of the uplink and downlink counters, which is closely associated with the cipher internal states. When the counter received in a new packet is exactly the expected value, the cipher will load the cipher information into the cipher hardware and generate LEN bits keystream to decrypt the received packet. If the MIC is valid, the local counter increases by 1 and the cipher internal state is saved. If ACK is required, the cipher for the other direction will be loaded and send an encrypted ACK back. This is the behavior of successful packet transmission.

When the counter in a received packet is not the expected value, it is known to the receiving node that one or several previous packets are lost. Assume the current counter is 0 and the received counter is 4. It is possible that 3 packets in between are lost. It is also possible that 19, 35, 51, ... packets are lost. Depending on the radio environment and the RSec system parameters, a cipher synchronization range should be set. For a regular network, the range from 0 to 16 seems to be reasonable.

To decrypt the received message, the cipher hardware is loaded with the cipher structure and the internal state corresponding to counter value 0. Then, a 3 times LEN-bit keystream is generated in order to move to the internal state corresponding to counter value 4. After that, a LEN bits keystream is produced to be XORed with the packet payload and MAC. MAC will be verified to see if the packet received is valid. If so, the cipher counter and internal states will be synchronized again. If not, there will be two possibilities: the received packet has a bit error during the transmission; or the counter should be moved forward by 16, which means 16 (4-bit counter) consecutive packets were lost prior to the new received packet.

To verify if it is the second case, the cipher will continue to generate a 15 LEN-bit keystream and use the 16th LEN-bit keystream to decrypt the message. If successful, the counter and internal states are synchronized again. If not, the message will be dropped, and the counter and internal state will not be updated. At this time, a local failure counter is set to 1, which means that one packet has failed on this cipher. If the counter reaches a threshold, which is assumed to be 3 here, the node will regard this cipher as out of synchronization and a resynchronization process is required.

3.3.7 Resynchronization

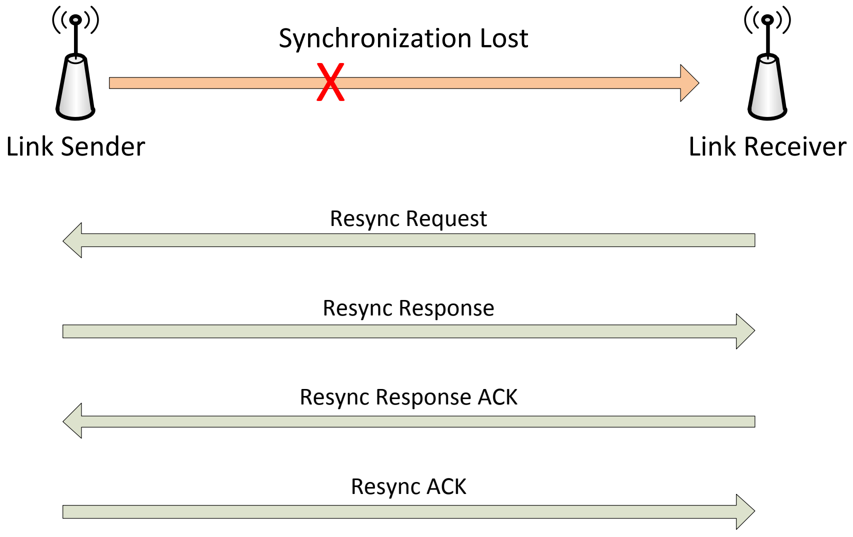

Figure 3.11: Resynchronization figure

In RSec communication systems, since two counters are managed for the uplink and the downlink, a node is both a sender and a receiver. When a cipher is out of synchronization, the resynchronization process is started, as shown in Fig. 3.11. This node as the link receiver will send an encrypted Resync Request message to its pair node, the link sender, to initialize the resynchronization process. The link sender node will generate a new 32-bit IV for this link and send it in the Resync Response message to the link receiver.

In the previous cipher information initialization phase, a secret key is saved by the two nodes in their local memory. The key will be used to generate the new cipher internal state. When the link receiver node gets the new IV from the Resync Response message, it loads the IV and the secret key to the cipher as the internal

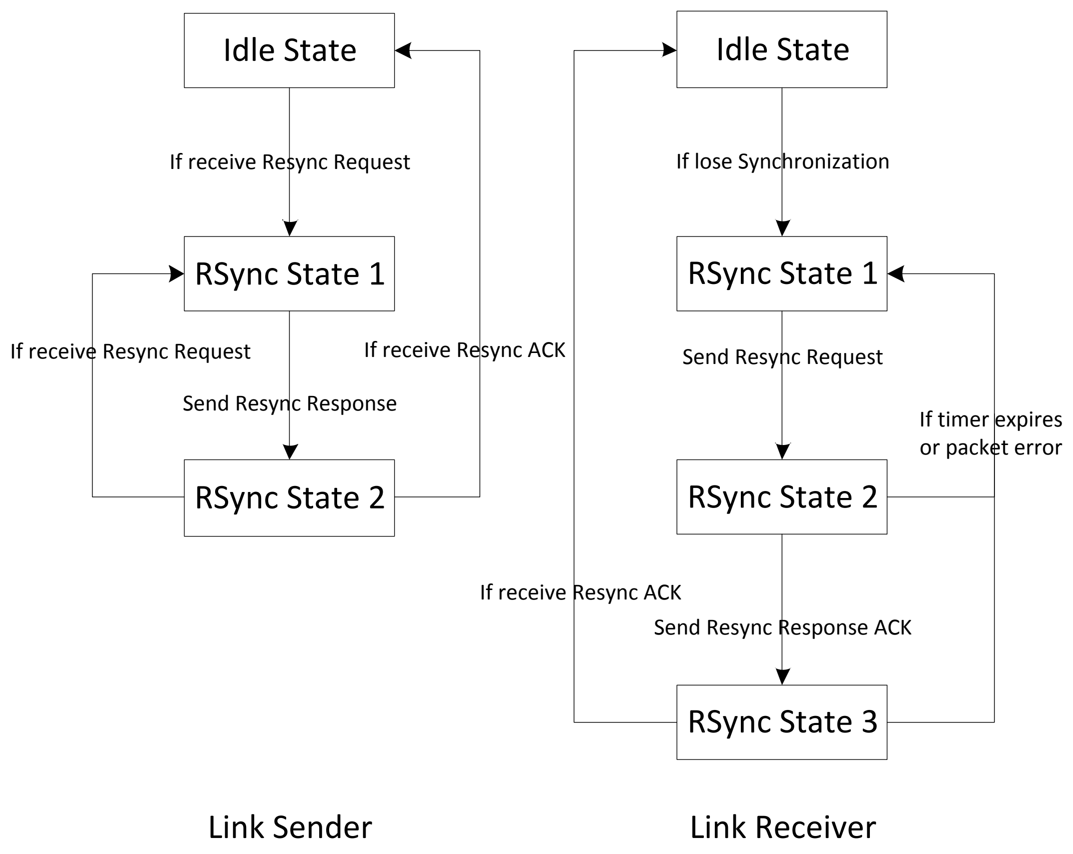

Figure 3.12: Resynchronization state machine

states, while the cipher structure remains the same. After cipher initialization, a LEN length keystream is produced to generate a Resync Response ACK to reply. At the same time, the local counter is reset to 0. The link receiver will receive the Resync Response ACK message and uses the same algorithm to verify if the link receiver has successfully updated the cipher internal state. If the verification is successful, the link sender will save the internal state and reset its local counter to 0. After that, a LEN-bit keystream is generated to encrypt a resync ACK which is sent to the link receiver. The new counter, which is increased from 0 to 1, is transmitted in this ACK message.

Since the resynchronization is likely caused by the interfering radio environment, it is possible that the resynchronization packets are lost too. The resync request is treated as a regular packet encrypted with the other link cipher, which is in synchronization. This packet requires a reply so a timer is started as soon as the packet is sent by the link receiver. If no reply is received when the timer expires, the request will be retransmitted with an increased counter.

If the resync request is received by the link sender, a resync response is replied. The link sender will not start a timer because if this packet is lost, the receiver will resend a resync request. If the resync response is received by the link receiver, a resync response ACK will be replied and a timer is started. If no packet is received when the counter expires or the received resync ACK fails in the validation, this resynchronization process is considered unsuccessful. Therefore, a new resync request will be sent. The current memory will be reset. The state machines of both nodes are shown in Fig. 3.12.

3.4 Analysis and Evaluation

The RAuth key management protocol and RSec link layer encryption architecture are introduced in the previous sections. Evaluations are carried out to test the performance of the proposed concepts. The link layer comparison focuses on the packet format overhead and packet loss performance. We also implement a network simulator to compare the authentication performance of the RAuth protocol with the LEAP protocol on the network initialization phase and new node joining phase.

3.4.1 RSec Link Layer Protocol Performance and Analysis

Previous research shows that the power consumption for a single bit transmission is equivalent to an embedded processor running several thousand of instructions [2]. Simplified packet formats can save a large amount of energy and extend the life period of a sensor node. TinyOS [53] is an embedded component-based operating system platform targeting WSNs. Based on the TinyOS link layer design, several packet formats were proposed in several link layer protocols, such as the TinySec-AE packet format of TinySec [54] and the MiniSec-U packet format of MiniSec [55].

Packet Formats

The packet formats of the mentioned protocols are shown in Fig. 3.13. In the RSec architecture, we do not implement the regular packet format defined in TinyOS to reduce the overhead. Careful choices of the fields in the packet format are made by taking the application requirements into consideration.

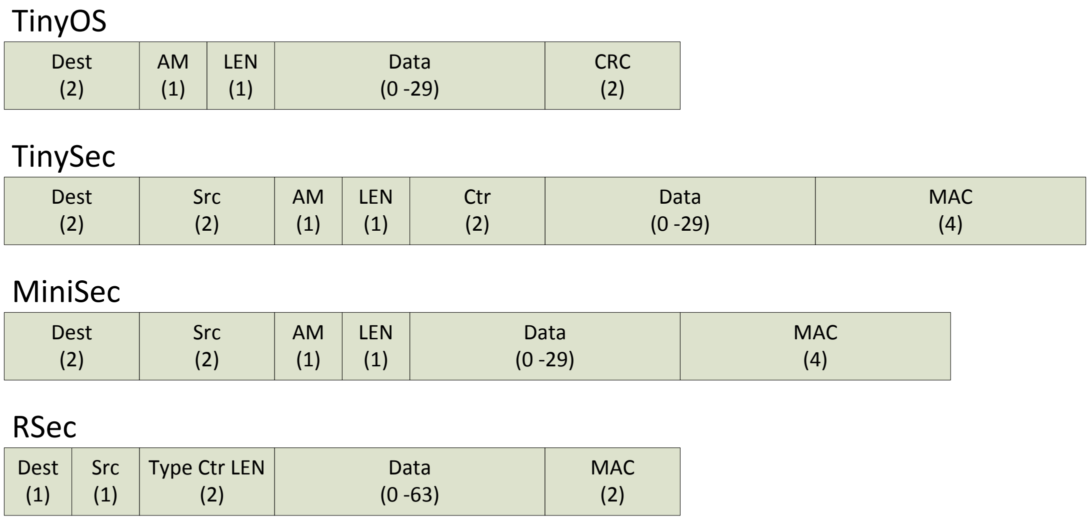
TinyOS
Dest (2) | AM (1) | LEN (1) | Data (0 -29) | CRC (2)
TinySec
Dest (2) | Src (2) | AM (1) | LEN (1) | Ctr (2) | Data (0 -29) | MAC (4)
MiniSec
Dest (2) | Src (2) | AM (1) | LEN (1) | Data (0 -29) | MAC (4)
RSec
Dest (1) | Src (1) | Type Ctr LEN (2) | Data (0 -63) | MAC (2)

Figure 3.13: The packet formats of TinyOS, TinySec, MiniSec and RSec

With the observation that the system performance drops significantly if a large number of nodes is applied, an RSec system limits the number of supported nodes in one network. In the RSec packet format, both the destination and the source addresses are 1 byte while the other protocols use 2 bytes. A maximum number of 256 addresses is supported in the RSec system while the others support up to 65536 addresses. If a large number of nodes are required in one application context, it is suggested to implement several RSec WSNs with different radio frequencies rather than a huge network with only one frequency. Different WSNs communicate with each other using the BSs as the gateway. Each BS forms a subnet of its own, and a node can only be accessed through the BS from outside. This practice is safe and efficient for the hierarchy network.

The type field of RSec has the same function as the active message (AM) field of the other protocols, which are used to indicate the message type. In the network simulation, only 14 types are used for the RAuth protocol and hence the 6-bit Type field with 64 maximum types is adequate for the RSec architecture.

As for the counter field, TinySec uses 2 bytes in the header to transmit the counter. In MiniSec, the 3 most significant bits (MSBs) of the LEN field are used to transmit the 3 last bits (LB) of the actual counter. TinySec uses the counter as part of the IV, and MiniSec uses the counter as the entire IV. In the RSec architecture, the counters are used to keep track of the cipher internal states.

The LEN fields are 8 bits in TinySec, 5 bits in MiniSec and 6 bits in RSec. Since the maximum length of the payload field in TinyOS is 29 bytes, the 5-bit LEN field is enough. RSec supports longer packet lengths to 63 bytes. The long payload is especially efficient in the cipher information assignment and update phase.

To minimize the transmit overhead, a 2-byte MAC is used in RSec while TinySec and MiniSec use 4 bytes. The MAC in RSec is the encrypted MIC while MIC guarantees the packet integrity, and the encrypted MAC guarantees the packet authenticity. MAC is efficient to detect transmission errors and defend against data forgery. In order to fake a 2-byte MAC, which has 65536 (2^{16}) possible values, the adversary has to try 32768 (2^{16-1}) times on average which is technically impossible. A 4-byte MAC will certainly provide stronger protection against the attacks. We consider a 2-byte MAC sufficient for the application context.

Table 3.1: Comparisons of Packet Formats

Protocols	RSec	TinySec	MiniSec
Header (Byte)	4	8	6
MAC (Byte)	2	4	4
Overhead (Byte)	6	12	10
Default Payload (Byte)	29	29	29
Max Payload (Byte)	63	29	29
Efficiency	82.9 %	70.7 %	74.3 %
Eff. Improvement	0	17.2 %	11.6 %

The comparisons on the packet format overhead are shown in Table 3.1. Since the TinyOS packet format fits limited application contexts, it is not listed in the comparison. Efficiency is the default payload size, 29 bytes, against the packet size, which is the sum of the default payload size and the overhead size. The last line shows the efficiency improvement of RSec against the other two protocols. From the results, we can see that the RSec architecture has better performance in the given application context.

Security Analysis

Security is one of the most important aspects to be considered in the link layer protocol. TinySec, MiniSec and RSec use different algorithms to guarantee the transmission confidentiality, integrity and authenticity.

TinySec uses the Skipjack block cipher in cipher block chaining (CBC) mode [56] as the cryptography algorithm to encrypt messages. CBC-MAC [57] is used to calculate the MAC. MiniSec uses a block cipher in offset codebook mode (OCB) [58] which encrypts the message and generates the corresponding MAC in one operation.

RSec uses an RFSR cipher for encryption and the CRC algorithm for integrity. The security analysis of RFSR is presented in the previous chapter. Unlike CBC-MAC and OCB algorithms, which directly generate the MAC, MAC in RSec is generated by encrypting the CRC checksums with the keystream. To verify the received message, the cipher information of the source is loaded corresponding to the message header. A keystream is produced to decrypt the message payload and MAC. Thereafter, the CRC algorithm is applied on the message header and the decrypted payload. If the checksum and the decrypted MAC are identical, the message is considered correct. An adversary cannot forge a message when he has no access to the keystream. Therefore, the MAC in RSec is considered secure to provide both integrity and authenticity.

3.4.2 Network Initialization

In this subsection, we compare the RAuth protocol with the LEAP protocol [34] on their authentication performance when a WSN initializes.

Simulation Environment and Settings

We focus on the situation where the WSN is used in a home area or a medium sized office. In these environments, we are assuming only one base station is in use. Some of the nodes have constant power supply which makes them a good choice to be the sink nodes. Some of the nodes that do not have constant power supply will be the regular sensor nodes.

Radio Platform Considering the radio platform with external power supply, the base station and the sink nodes do not have energy constraints on the radio module. They can choose to use maximum TX output power to increase coverage and improve the packet transmission quality. Besides, a better receive sensitivity can be utilized to improve the signal to noise ratio (SNR) to decrease the packet error rate (PER) in transmission. For the sensor nodes, since the internal power is limited, in order to extend the battery life, the radio module on a sensor node works in a relatively lower power consuming status. The transmit power is lower and the receiving sensitivity is higher to save energy. Since the PER will go up rapidly when the transmission power drops, to ensure the communication link with high quality, a minimum receivable power (minrcvp) is used as a threshold for a node to determine whether to establish a link with the other node based on the received signal strength. For the cases when the received signal strength is below the threshold, even though it is not good enough to establish a link, the receiver can actually sense the signal and avoid transmitting at the same time. To avoid packet collision, the sensing power (sensp) is utilized for the radio module to decide if the channel is occupied in the current time slot.

As sensor networks draw more and more attention from academy and industry areas, various hardware platforms are designed for testing and deployment, such as the Mica series, Telos motes and so on [59]. The radio modules are mostly designed for very low-power wireless communications. These radio modules usually work at 2.4 GHz or Sub-1 GHz ISM bands. Take TI CC1100 as an example. It works at 315/ 433/ 915 MHz with data rates of 1.2 to 500 kbps. The output power range is from -30 dBm to 10 dBm. CC1100 also supports low power listening functionality. To simulate a sensor network properly, we use the following parameters shown in Table 3.2.

Table 3.2: Radio parameters used in simulation.

Node Types	BS	Sink	Sensor Node
TX Power (dBm)	4	4	-20
Min. Receivable Power (dBm)	-80	-80	-70
Min. Sensible Power (dBm)	-100	-100	-80

Topology We place the simulation in a square open field of 24 by 24 meters as a 2 dimensional coordinate system, which is shown in Fig. 3.14. Both the x and y axses are in the range of 1 to 25. In order to test the coverage and signal strength on the edge, we set 81 nodes evenly in the field. Their x-axis and y-axis positions are from 1 to 25 with the separation distance of 3. In order to add some uncertainness, 19 nodes are sprayed in the field randomly. The base station is chosen at position (9, 12), which is near the center of the field. Four sink nodes are chosen not far from the center of each quarter of the field, at position (7, 22), (7, 4), (19, 19) and (16, 7).

CSMA/CA is utilized to avoid packet collisions. A node will listen to the channel and wait for a vacancy. Then the node will broadcast a channel occupation message which specifies the receiver node. When the message is received by the destination node, it will also broadcast a channel occupation message to notify its neighbors of

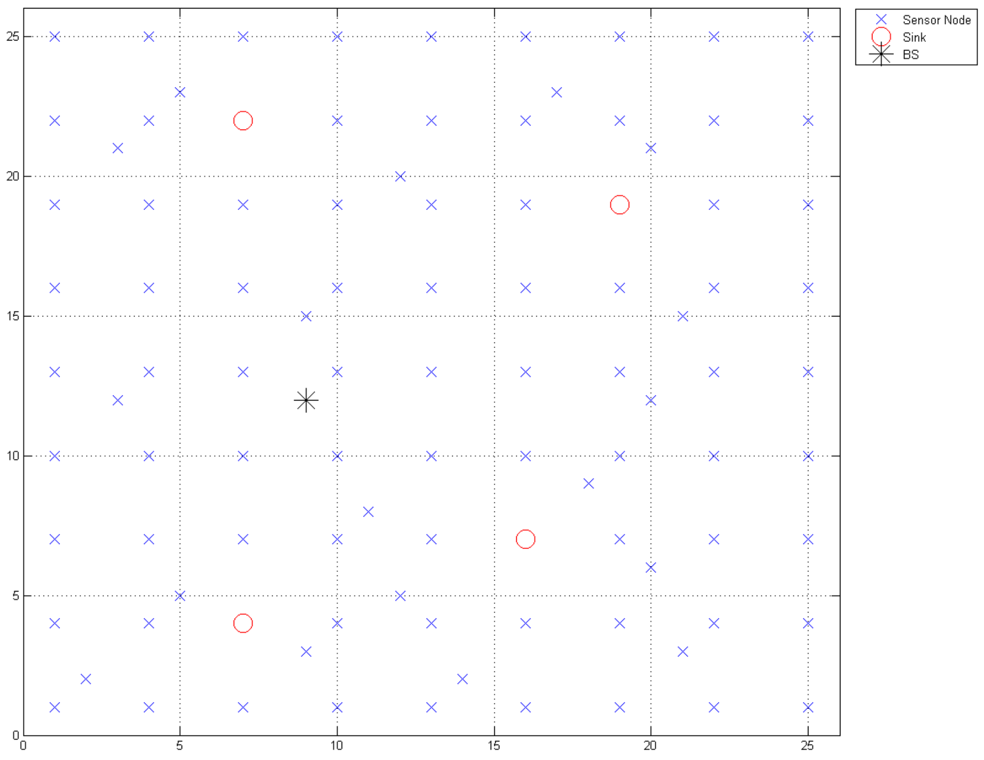

Figure 3.14: Simulation topology

an upcoming transmission. After all these, the sender will send the packet to the destination. This technique efficiently avoids packet collisions.

To simplify the simulation and only focus on the differences brought by different authentication protocols, several assumptions are made. Before the authentication process starts, all the nodes are already deployed and ready to send or receive packets. For the LEAP scenario, all the nodes will start their authentication broadcasting at the same time. For the RSec scenario, the nodes are in the active mode so the authentication only starts when the nodes receive an authentication broadcast challenge message from the base station or the sinks. Besides, since the minimum receivable power is used to ensure the communications' quality, it is considered that no transmission error will occur on a communication link. In the RSec scenario, one sensor node may receive an authentication broadcast message from the base station and several sinks, to ensure the connection quality, it will choose the one with lowest path loss as its parent node.

As for the time dimension, we split time into small time slots. The time slot is the minimum time unit which guarantees that a packet can be transmitted within. The channel occupation messages and the ACKs are considered within the same time slot of the packet transmission.

Channel Path Loss Model Sensor nodes usually have limited output power. When the received signal strength is low, the probability of error will increase. Therefore, to make the simulation more accurate, the signal strength should be taken into consideration. One of the most common radio propagation models is the log-normal shadowing path loss model [60]. This model can be used for large and small coverage systems [61]. Empirical studies [62] have shown that the log-normal shadowing model provides more accurate multi-path channel models than Nakagami and Rayleigh for indoor environments. The model is given by:

$$PL(d) = PL(d_0) + 10nlog_{10}(d/d_0) + X_\sigma \tag{3.1}$$

where d is the distance between the transmitter and receiver, d_0 a reference distance, n the pass loss exponent, and X_σ a zero-mean Gaussian random variable with standard deviation σ. To simplify the simulation, we do not consider the shadowing effects.

Besides, a previous study [63] measured the channel behavior in the 800 - 1000 MHz band in different environments. From the descriptions of the environment, we choose the parameters in Table 3.3 which best match the application context.

Table 3.3: Log-normal shadowing model parameters.

n	$PL(d_0)$	X_σ
2.0	36.6	0

LEAP Scenario

At time slot 0, all the nodes will start broadcasting their HELLO message. Since the channel can only be accessed by one node at one time slot, the simulation system will have to randomly allocate the channel to one node. Each node maintains a task list in its memory of the outgoing packets. When the channel is allocated to it by the simulation system, the first message in the task list will be popped out and transmitted. It is possible that several packets are transmitted in the same time slot, but only if all the senders and the receivers are physically separated by long distances that one transmission will not be sensed by the other nodes.

$$\begin{aligned} u &\longrightarrow * : \quad u. \\ v &\longrightarrow u : \quad v, \{K_v^c\}_{K_v}, MAC(K_v, u|v|\{K_v^c\}_{K_v}). \\ u &\longrightarrow v : \quad u, \{K_u^c\}_{K_{uv}}, MAC(K_v, u|\{K_u^c\}_{K_{uv}}). \end{aligned}$$

Figure 3.15: LEAP key establishment procedure

Following the LEAP scenario, a three-way handshake is performed as shown in Fig. 3.15. First, the new node u will broadcast its own id. When the neighbor node v receives the broadcast message, it will reply to this broadcast message. Thereafter, the node u will respond to this message to finish the authentication process and establish a communication link.

In the simulation, when node u receives node v's authentication message and also v's authentication broadcast message, node u will ignore v's authentication broadcast message and will only reply to the authentication message. All nodes in the LEAP simulation scenario are considered regular sensor nodes.

RSec Scenario

Similar to the LEAP scenario, each node maintains a task list in its memory and tries to occupy the channel to send packets. However, the RSec scenario introduces the message priority mechanism which makes the message with higher priority sent in the channel earlier to improve the system performance. The priority mechanism can be achieved in real systems by setting different interframe spacing (IFS) based on the priority, which is efficiently used in the 802.11 MAC layer protocol. The packet with higher priority has shorter IFS. Until the channel is not occupied, the nodes have to wait for a certain time before trying to occupy the channel. The channel will be taken over by the node with the shortest IFS.

As mentioned, the active mode is used in the simulation for RSec systems. At time slot 0, the base station will start the whole process by broadcasting the authentication challenge message. When the sinks receive the challenge message from the base station, they will first append the authentication response messages to their task lists, and then append the sink broadcasting challenge messages to the task lists.

After that, the sensor nodes will start the authentication process when challenge messages are received.

The tasks in the task list can be changed when they are not sent. When a sensor node receives the first authentication challenge message, it will append a response message to the task list. Some time slots later, when the sensor node receives another authentication challenge message that has a different link budget, the sensor node will choose the sink or the base station with a higher link budget than its parent node. If needed, it will correct the authentication challenge response message in its task list.

Simulation Results and Analysis

The simulation is carried out using the scenarios stated previously. The simulation finishes when all the task lists are empty. The simulation result is shown in Table 3.4. The total time indicates the number of time slot is used when all the nodes finish the authentication; the number of messages is the total number of authentication messages that are sent/received in the whole authentication process.

Table 3.4: Simulation results of RAuth and LEAP.

Protocol	Total Time	Num of Msg.
RAuth	365	365
LEAP	461	962
Improvement	20.1%	62.1%

The total time and the number of messages of RAuth are both 365, which indicates that only one message is sent in each time slot. Since CSMA/CA is employed by the nodes, before each transmission, the sender will broadcast an RTS message and the receiver will broadcast a CTS message. Because RAuth is a hierarchy centralized network, at least one of the two nodes in each transmission will be the BS or the Sink, which has large TX power so that most nodes can sense the on-going transmission. On the other hand, LEAP implements with flat topology where nodes only authenticate with their neighbors. It is possible that multiple pairs of nodes are transmitting at the same time slot.

In the LEAP simulation scenario, each node will broadcast the HELLO message. Therefore, 100 HELLO messages are sent in total. The total number of messages transmitted except the HELLO messages is 862. LEAP utilizes a 3-way handshake authentication approach in which the HELLO message is the first one. Therefore, 431 node pairs are established in total. On average, each node has 4.31 nodes as its neighbors.

An important point should be addressed: after authentication, the nodes in the RAuth scenario are equipped with the new pairwise cipher information so that the communication can start right away. However, even though the nodes in the LEAP scenario also establish the pairwise keys, they have no idea about the nodes outside of their neighbor. An additional routing protocol will be started to help those nodes establish local routing tables. The routing table update process will take longer than the authentication which will extend the network initialization time.

Compared with the LEAP protocol, the RAuth protocol has better performance in the network initialization phase. The total time used for authentication is 20.1% shorter than the LEAP protocol, and the total number of message is 62.1% less than LEAP protocol.

After time T_{min} of deployment, according to the LEAP protocol, the initial key, which help sensor nodes establish pairwise keys, will be deleted from the node. LEAP assumes that no adversary can successfully derive the initial key from any node. This assumption is weak because if the estimation of T_{min} is inaccurate or a new efficient attack is employed, the LEAP network will be jeopardized. With the initial key, the adversary can establish a new pairwise key with any node in the network within or out of the time T_{min}. All the communications in the network can be eavesdropped and decrypted.

Regarding the RAuth protocol, even though a node is compromised by the adversary, only the cipher information related to this node is revealed. If a sensor node is compromised, the adversary gets the cipher information about the node with its parent, the node with the base station. If a sink is compromised, even though the adversary has some cipher information, the traffic, which neither starts nor ends at the this sink node, is still encrypted with another cipher unknown to this sink. Therefore, RAuth is an efficient yet secure protocol.

3.4.3 Conclusion

In this chapter, we have proposed the RFSR cipher based secure authentication protocol RAuth and the secure link layer architecture RSec. Comparing with TinySec and MiniSec protocols, RSec has 17.2 % and 11.6 % improvement respectively. The network initialization with RAuth and LEAP protocols is simulated, where the RAuth protocol outperforms the LEAP protocol. RAuth uses 20.1% less total time and 62.1% fewer total messages to finish a 100-node network authentication initialization. In terms of total initialization time, RAuth outperforms LEAP. A security analysis is made which proves that the RAuth protocol is more secure than LEAP.

Chapter 4

Conclusions and Future Work

4.1 Conclusions

In this thesis, we have introduced a security solution including a cipher, an authentication protocol and a link layer encryption framework.

In Chapter 2, we have proposed a low complexity reconfigurable feedback shift register based stream cipher RFSR and shown that it is more secure than the widely used Grain, RC4 and RC5 cryptography algorithms. Implemented on an FPGA platform, the proposed scheme consumes over 130 times less average energy, and renders over 25 times less delay than existing microprocessor platforms.

In Chapter 3, we have proposed the RFSR cipher based secure authentication protocol RAuth and the secure link layer architecture RSec. Comparing with TinySec and MiniSec protocols, RSec has 17.2 % and 11.6 % improvement, respectively. The network initialization with the RAuth and the LEAP protocols is simulated, where the RAuth protocol outperforms the LEAP protocol. RAuth uses 20.1% less total time and 62.1% less total messages to finish a 100-node network authentication initialization. In the perspective of total initialization time, RAuth outperforms LEAP. A security analysis is made which proves that the RAuth protocol is more secure than LEAP.

4.2 Future Work

The proposals are analyzed based on the simulation results. For the future work, the system can be prototyped and tested in the real environment. The cipher can be implemented and downloaded to FPGA and the actual power consumption should be measured. Several prototype sensor nodes can be implemented with the RAuth protocol and the RSec framework.

Appendix A

Network Initialization Simulation Code

As discussed in Section 3.4.2, the network initialization phase is simulation with Python implementation. The whole project consists of four source files which are listed below.

main.py

```
import sys, math
from lib import Msg,Node,RfsrNode,LeapNode
from globals import ctime,lgr,lgr1,node,node1,random,prior

def topology_load(TopoFileHandler,rfsr = True):
    TopoFile = open("topology_file",'r')
    for line in TopoFile.readlines():
        if(rfsr):
            NewNode = RfsrNode(line.split()[1:],int(line.split()[0]))
            node.append(NewNode)
            lgr.debug(NewNode.__dict__)
        else:
            NewNode = LeapNode(line.split()[1:],int(line.split()[0]))
            node1.append(NewNode)
            lgr1.debug(NewNode.__dict__)
    TopoFile.close()

def RfsrResult():
    global node
    for n in node:
        if n.type == 'Base':
            lgr.debug('')
            lgr.debug('id %d, num: %d' % (n.id,len(n.permitlist)))
            for i in n.permitlist:
                lgr.debug('%-3d %-3.1f' % (i,n.sig[i]))
        elif n.type == 'Sink':
            lgr.debug('')
            lgr.debug('id %d, num: %d' % (n.id,len(n.permitlist)))
            for i in n.permitlist:
                lgr.debug('%-3d %-3.1f' % (i,n.sig[i]))
        elif n.type == 'Node':
            if (n.nexthop == []):
                lgr.debug('')
                lgr.debug('id %d' % n.id)

def LeapResult():
    global node1
    for n in node1:
        for nnid in n.authlist:
            if n.id not in node1[nnid-1].authlist:
                lgr.debug('')
                lgr.debug('%2d in %2d authlist, but not way around' % (
   nnid, n.id))
    for n in node1:
        n.printauthlist()

def powercalc(src,dst, is_rcv=True):
    dist=((src.x-dst.x)**2+(src.y-dst.y)**2)**0.5
    pl = 36.6 + 10*2.0*math.log10(dist)
    if is_rcv:
        return (src.sendp-dst.minrcvp-pl)
    else:
        return (src.sendp-dst.minrcvp-pl, src.sendp-dst.sensp-pl)

def AdjNodeFind(node):
    # AdjList [ (NodeID, dist), (NodeID, dist), ... ]
    def add(n1, n2):
        rx_t, rx_s = powercalc(n1,n2,False)
        #print("node %d %d %.1f %.1f" % (n1.id,n2.id,rx_t,rx_s))
        if (rx_t >= 0):
            n1.tlist.append(n2.id)
        elif (rx_s >= 0):
            n1.slist.append(n2.id)
    for n in node:
        for nn in node:
            if (nn is not n):
                add(n,nn)

def RfsrMain():
    global ctime
    global node
    TaskList = []
    # avalist uses node.id
    avalist = list(range(1,len(node)+1))
    for n in node:
        if(not n.tlempty()):
            TaskList.append(n.id)
            lgr.debug('id %d' % n.id)
            n.printtasklist()
    if (TaskList == []):
        lgr.debug('TaskList is empty, exit')
        return False
    lgr.debug('')
    lgr.debug('#############')
    lgr.debug('#time is %-3d#' % ctime[0])
    lgr.debug('#############')
    while (TaskList):
        lgr.debug('')
        lgr.debug('TaskList is: %s' % str(TaskList))
        for i in prior:
            tmplist = []
            found = False
            for t in TaskList:
                for tt in node[t-1].tasklist:
                    if tt.fn==i:
                        tmplist.append((t,node[t-1].tasklist.index(tt)))
                        found = True
            if(found):
                break
        lgr.debug('tmplist with prior %d is %s' % (i,str(tmplist)))
        node_id,taskid = tmplist[random.randint(0,len(tmplist)-1)]
        lgr.debug('random node.id: %d' % node_id)
        thenode = node[node_id-1]
        msg = thenode.getmsg(taskid)
        msg_ok = True
        for node_id1 in thenode.tlist+thenode.slist:
            if(node_id1 not in avalist):
                msg_ok = False
                break
        if(not msg_ok):
            TaskList.remove(node_id)
            lgr.debug('node.id %d has conflict receiving node' % node_id)
            continue
        else:
            thenode.popmsg(taskid)
            TaskList.remove(node_id)
        lgr.debug('node.id is %d, msg is %s' % (node_id,msg.printmsg()))
        if (msg.dst==999):
            for adjnode_index in thenode.tlist:
                # node index is 0-based
                # node id is 1-based
                node[adjnode_index-1].rcv(msg,powercalc(thenode,node[adjnode_index-1]))
        else:
            node[msg.dst-1].rcv(msg,powercalc(thenode,node[msg.dst-1]))
            rcvnodeid = msg.dst
            for n in node[rcvnodeid-1].tlist+node[rcvnodeid-1].slist:
                if n in TaskList:
                    TaskList.remove(n)
                if n in avalist:
                    avalist.remove(n)
        for n in thenode.tlist+thenode.slist:
            if n in TaskList:
                TaskList.remove(n)
            if n in avalist:
                avalist.remove(n)
    return True

def LeapMain():
    global ctime
    TaskList = []
    # avalist uses node.id
    avalist = list(range(1,len(node1)+1))
    for n in node1:
        if(not n.tlempty()):
```

```python
            TaskList.append(n.id)
    if (TaskList == []):
        lgr1.debug('TaskList is empty, exit')
        return False
    lgr1.debug('')
    lgr1.debug('#############')
    lgr1.debug('#time is %-3d#' % ctime[1])
    lgr1.debug('#############')
    while (TaskList):
        lgr1.debug('')
        lgr1.debug('TaskList is: %s' % str(TaskList))
        node_id = TaskList[random.randint(0,len(TaskList)-1)]
        lgr1.debug('random node.id: %d' % node_id)
        thenode = node1[node_id-1]
        msg = thenode.getmsg()
        msg_ok = True
        if (msg.dst == 999):
            for node_id1 in thenode.tlist+thenode.slist:
                if(node_id1 not in avalist):
                    msg_ok = False
                    break
        elif (msg.dst not in avalist):
            msg_ok = False
        if(not msg_ok):
            TaskList.remove(node_id)
            lgr1.debug('node.id %d has conflict receiving node' % node_id)
            continue
        else:
            thenode.popmsg()
        lgr1.debug('node.id is %d, msg is %s' % (node_id,msg.printmsg()))
        if (msg.dst==999):
            for adjnode_index in thenode.tlist:
                # node index is 0-based
                # node id is 1-based
                if msg.src != adjnode_index:
                    node1[adjnode_index-1].rcv(msg,powercalc(thenode,node1[adjnode_index-1]))
        else:
            node1[msg.dst-1].rcv(msg,powercalc(thenode,node1[msg.dst-1]))
        for n in thenode.tlist+thenode.slist:
            if n in TaskList:
                TaskList.remove(n)
            if n in avalist:
                avalist.remove(n)
        if thenode.id in avalist:
            avalist.remove(thenode.id)
        if thenode.id in TaskList:
            TaskList.remove(thenode.id)
    return True

def rfsr_run():
    # RFSR gogogo
    # load topology file
    lgr.debug('##########################')
    lgr.debug('###Node loading started###')
    lgr.debug('##########################')
    topology_load("topology")
    if (not node):
        lgr.debug("topology load failure")
        sys.exit()

    lgr.debug('##############')
    lgr.debug('###Find Adj###')
    lgr.debug('##############')
    AdjNodeFind(node)
    for n in node:
        n.printnodeall()

    lgr.debug('################')
    lgr.debug('###Start Auth###')
    lgr.debug('################')
    for n in node:
        n.authInit()

    while(RfsrMain() and ctime[0] < 10000):
        ctime[0] += 1

    RfsrResult()

def leap_run():
    # LeapNode Main function
    lgr1.debug('##########################')
    lgr1.debug('###Node loading started###')
    lgr1.debug('##########################')
    topology_load("topology",False)
    if (not node1):
        lgr1.debug("topology load failure")
        sys.exit()

    lgr1.debug('##############')
    lgr1.debug('###Find Adj###')
    lgr1.debug('##############')
    AdjNodeFind(node1)
    for n in node1:
        n.printnodeall()

    lgr1.debug('################')
    lgr1.debug('###Start Auth###')
    lgr1.debug('################')
    for n in node1:
        n.authInit()
    while(LeapMain() and ctime[1]<1000):
        ctime[1] += 1
        for n in node1:
            if(not n.tlempty()):
                n.printnode()

    LeapResult()

leap_run()
rfsr_run()
print("#################")
rfsr_send = 0
leap_send = 0
leap_neighbour = 0
rfsr_leftnode = []
leap_link = 0
for n in node:
    rfsr_send += n.sendcnt
    if (n.type == "Node"):
        if(n.nexthop == 999):
            rfsr_leftnode.append(n.id)
for n in node1:
    leap_send += n.sendcnt
    leap_link += len(n.authlist)
print("rfsr_send %d" % rfsr_send)
print("leap_send %d" % leap_send)
if (len(rfsr_leftnode)):
    print("rfsr not auth node")
    print(rfsr_leftnode)
print("leap total links: %d" % leap_link)
print("rfsr_time %d" % ctime[0])
print("leap_time %d" % ctime[1])

print("DONE")
```

lib.py

```
from globals import ctime,env_power,lgr,lgr1

class Msg(object):
    """MAC Layer message"""
    def __init__(self, dst, src, fn, time, msg=None):
        self.dst = dst
        self.src = src
        self.fn  = fn
        self.time = time
        self.msg = msg
    def printmsg(self):
        return(str(self.__dict__))

class Node(object):
    """Sensor Node Class"""
    def __init__(self, args, nid):
        self.x     = int(args[0])
        self.y     = int(args[1])
        if (args[2]=='0'):
            self.type = 'Node'
        elif (args[2]=='1'):
            self.type = 'Base'
        else:
            self.type = 'Sink'
        # self.type  = args[2]
        #1-based id
        self.id    = nid+1
        # transmission list, node.id in this list can rcv my msg
        self.tlist = []
        # sensing list, node.id in this list can only sense my msg
        self.slist = []
        self.tasklist = []
        self.sig = {}
        self.sendcnt = 0
    def tlempty(self):
        if (self.tasklist == []):
            return True
        else:
            return False
    def getmsg(self,taskid=0):
        return(self.tasklist[taskid])
    def popmsg(self,taskid=0):
        self.tasklist.pop(taskid)
        self.sendcnt += 1

class RfsrNode(Node):
    """Node running RFSR auth protocol"""
    def __init__(self, args, nid):
        Node.__init__(self, args, nid)
        # send power, min receive power and min sense power
        self.sendp   = env_power["sendp"][self.type]
        self.minrcvp = env_power["minrcvp"][self.type]
        self.sensp   = env_power["sensp"][self.type]
        if (self.type == 'Base'):
            self.route = {}
            self.permitlist = []
        elif (self.type == 'Sink'):
            self.bs = 999
            self.permitlist = []
        else:
            self.bs = 999
            self.nexthop = 999
    def authInit(self,asked=False):
        global ctime
        if (self.type == 'Base'):
            self.tasklist.append(Msg(999,self.id,0,ctime[0]))
        else:
            if(self.type == 'Sink'):
                self.tasklist.append(Msg(999,self.id,5,ctime[0]))
            else:
                fn = 20
    def sinkfwd2base(self, dst, msg):
        global ctime
        need_newmsg = True
        for task in self.tasklist:
            if (task.fn==7):
                task.msg.append(msg)
                need_newmsg = False
        if (need_newmsg):
            self.tasklist.append(Msg(dst,self.id,7,ctime[0],[msg]))
    def nodereplysink(self,sinkid):
        global ctime
        need_newmsg = True
        for task in self.tasklist:
            if (task.fn==6 or task.fn==3):
                if (self.sig[task.dst]<=self.sig[sinkid]):
                    task.dst = sinkid
                    if (task.fn ==3):
                        task.fn = 6
                        task.msg = [self.id,self.bs]
                    task.time = ctime[0]
                need_newmsg = False
        if (need_newmsg):
            self.tasklist.append(Msg(sinkid,self.id,6,ctime[0],[self.id,
self.bs]))
    def rcv(self, msg, sig):
        global ctime
        # Base
        if(sig<0):
            return
        if(self.type == "Base"):
            if(msg.dst==self.id):
                self.sig[msg.src] = sig
                if(msg.fn==1):
                    # verify sink identity
                    if(msg.msg==msg.src):
                        # TODO: append new cipher info
                        self.tasklist.append(Msg(msg.src,self.id,2,ctime
[0],msg.src+self.id))
                        self.route[msg.src] = msg.src
                        self.permitlist.append(msg.src)
                elif(msg.fn==3):
                    # verify node identity
                    if(msg.msg==msg.src):
                        # TODO: append new cipher info
                        self.tasklist.append(Msg(msg.src,self.id,4,ctime[0],msg.src+self.id))
                        self.route[msg.src] = msg.src
                        self.permitlist.append(msg.src)
                elif(msg.fn==7):
                    for amsg in msg.msg:
                        if amsg[1]==999:
                            # TODO: verify node certification
                            self.tasklist.append(Msg(msg.src,self.id,8,ctime[0],amsg[0]))
                            self.route[amsg[0]]=msg.src
                        elif amsg[1]==self.id:
                            self.tasklist.append(Msg(amsg[0],self.id,10,ctime[0],['yes',msg.src]))
                            self.route[amsg[0]]=amsg[0]
                            self.tasklist.append(Msg(msg.src,self.id,11,ctime[0],['yes',amsg[0]]))

        # Sink
        if(self.type == "Sink"):
            self.sig[msg.src] = sig
            if(msg.dst==999):
                if(msg.fn==0):
                    self.bs = msg.src
                    self.tasklist.append(Msg(msg.src,self.id,1,ctime[0],self.id))
            elif(msg.dst==self.id):
                if(msg.fn==2):
                    # verify server identity
                    if(msg.msg==self.id+msg.src):
                        self.permitlist.append(msg.src)
                if(msg.fn==6):
                    self.sinkfwd2base(self.bs, msg.msg)
                if(msg.fn==8):
                    # TODO: renew cipher
                    self.tasklist.append(Msg(msg.msg,self.id,9,ctime[0],'yes'))
                    self.permitlist.append(msg.msg)
                if(msg.fn==11):
                    self.permitlist.append(msg.msg[1])
                    self.tasklist.append(Msg(msg.msg[1],self.id,12,ctime[0],'handshake'))

        # Node
        if(self.type == "Node"):
            self.sig[msg.src] = sig
            if(msg.dst==999):
                if(msg.fn==0):
                    self.bs = msg.src
                    self.tasklist.append(Msg(msg.src,self.id,3,ctime[0],self.id))
                if(msg.fn==5):
                    #if(self.bs!=self.nexthop or (self.bs==999 and self.nexthop==999)):
                    self.nodereplysink(msg.src)
            elif(msg.dst==self.id):
                if(msg.fn==4):
                    # verify server identity
                    if(msg.msg==self.id+msg.src):
                        self.nexthop = msg.src
                if(msg.fn==9):
                    if(msg.msg=='yes'):
                        self.nexthop=msg.src
                if(msg.fn==10):
                    if(msg.msg[0]=='yes'):
                        self.nexthop=msg.msg[1]
    def printnodeall(self):
        lgr.debug('')
        self.printnodebasic()
    def printnodebasic(self):
        # print x, y, tlist, slist
        lgr.debug('id:%2d, x:%2d y:%2d' % (self.id,self.x,self.y))
        lgr.debug('tlist: %s' % str(self.tlist))
        lgr.debug('slist:%s' % str(self.slist))
    def printtasklist(self):
        for msg in self.tasklist:
            lgr.debug(msg.printmsg())
    def printpower(self):
        lgr.debug('powerlist:%s' % str(self.sig))

```

```python
class LeapNode(Node):
    """Node running LEAP auth protocol"""
    def __init__(self, args, nid):
        Node.__init__(self, args, nid)
        self.authlist = []
        self.sendp   = env_power["sendp"]["Node"]
        self.minrcvp = env_power["minrcvp"]["Node"]
        self.sensp   = env_power["sensp"]["Node"]
    def printauthlist(self):
        lgr1.debug('')
        lgr1.debug('id %d' % self.id)
        lgr1.debug('authlist:%s' % str(self.authlist))
    def printtasklist(self):
        for msg in self.tasklist:
            lgr1.debug(msg.printmsg())
    def printnodebasic(self):
        # print x, y, tlist, slist
        lgr1.debug('id:%2d, x:%2d y:%2d' % (self.id,self.x,self.y))
        lgr1.debug('tlist: %s' % str(self.tlist))
        lgr1.debug('slist: %d' % len(self.slist))
    def printnode(self):
        lgr1.debug('')
        lgr1.debug('id %d' % self.id)
        self.printtasklist()
    def printnodeall(self):
        lgr1.debug('')
        self.printnodebasic()
    def authInit(self):
        global ctime
        self.tasklist.append(Msg(999,self.id,0,ctime[1]))
    def authRemoveduplicate(self, nodeid):
        for msg in self.tasklist:
            if(msg.dst == nodeid):
                self.tasklist.remove(msg)
    def rcv(self, rcvmsg, sig):
        global ctime
        if (sig<0):
            return
        if(rcvmsg.dst==999 and rcvmsg.fn==0):
            # rcv broadcast auth request
            if (rcvmsg.src not in self.authlist):
                self.tasklist.append(Msg(rcvmsg.src,self.id,1,ctime[1],'
    ok'))
        elif(rcvmsg.dst==self.id and rcvmsg.fn==1):
            # auth success
            self.authlist.append(rcvmsg.src)
            self.authRemoveduplicate(rcvmsg.src)
            lgr1.debug('node.id %2d add %2d to authlist' % (self.id,
    rcvmsg.src))
            self.tasklist.append(Msg(rcvmsg.src,self.id,2,ctime[1],'ok')
    )
        elif(rcvmsg.dst==self.id and rcvmsg.fn==2):
            self.authlist.append(rcvmsg.src)
            self.authRemoveduplicate(rcvmsg.src)
            lgr1.debug('node.id %2d add %2d to authlist' % (self.id,
    rcvmsg.src))
        else:
            lgr1.debug("!!! node.id %2d received wrong msg %s" % (self.
    id,rcvmsg.printmsg()))
```

globals.py

```
import logging
import random

lgr = logging.getLogger('nsim-rfsr')
lgr.setLevel(logging.DEBUG)
# add a file handler
fh = logging.FileHandler('dbg-rfsr.log', mode='w')
fh.setLevel(logging.DEBUG)
# create a formatter and set the formatter for the handler.
frmt = logging.Formatter('%(message)s')
fh.setFormatter(frmt)
# add the Handler to the logger
lgr.addHandler(fh)

lgr1 = logging.getLogger('nsim-leap')
lgr1.setLevel(logging.DEBUG)
# add a file handler
fh1 = logging.FileHandler('dbg-leap.log', mode='w')
fh1.setLevel(logging.DEBUG)
# create a formatter and set the formatter for the handler.
frmt1 = logging.Formatter('%(message)s')
fh1.setFormatter(frmt1)
```

```
# add the Handler to the logger
lgr1.addHandler(fh1)

# Global Variables
node = []
node1 = []
ctime = [0,0]
env_power = {
    "sendp" : {  'Base': 4,
                 'Sink': 4,
                 'Node': -20},
    "minrcvp" : {'Base': -80,
                 'Sink': -80,
                 'Node': -70},
    "sensp" : {  'Base': -100,
                 'Sink': -100,
                 'Node': -80}
}
prior = [0,2,1,8,10,11,4,5,3,9,12,6,7]

# random
random.seed(9)
```

topology_file

```
0   1   1   0
1   1   4   0
2   1   7   0
3   1   10  0
4   1   13  0
5   1   16  0
6   1   19  0
7   1   22  0
8   1   25  0
9   4   1   0
10  4   4   0
11  4   7   0
12  4   10  0
13  4   13  0
14  4   16  0
15  4   19  0
16  4   22  0
```

```
17  4   25  0
18  7   1   0
19  7   4   2
20  7   7   0
21  7   10  0
22  7   13  0
23  7   16  0
24  7   19  0
25  7   22  2
26  7   25  0
27  10  1   0
28  10  4   0
29  10  7   0
30  10  10  0
31  10  13  0
32  10  16  0
33  10  19  0
34  10  22  0
35  10  25  0
36  13  1   0
37  13  4   0
38  13  7   0
39  13  10  0
40  13  13  0
41  13  16  0
42  13  19  0
43  13  22  0
44  13  25  0
45  16  1   0
46  16  4   0
47  16  7   2
48  16  10  0
49  16  13  0
50  16  16  0
51  16  19  0
52  16  22  0
53  16  25  0
54  19  1   0
55  19  4   0
56  19  7   0
57  19  10  0
58  19  13  0
```

```
59  19  16  0
60  19  19  2
61  19  22  0
62  19  25  0
63  22  1   0
64  22  4   0
65  22  7   0
66  22  10  0
67  22  13  0
68  22  16  0
69  22  19  0
70  22  22  0
71  22  25  0
72  25  1   0
73  25  4   0
74  25  7   0
75  25  10  0
76  25  13  0
77  25  16  0
78  25  19  0
79  25  22  0
80  25  25  0
81  5   23  0
82  17  23  0
83  3   21  0
84  12  20  0
85  20  21  0
86  9   15  0
87  21  15  0
88  3   12  0
89  9   12  1
90  20  12  0
91  11  8   0
92  18  9   0
93  20  6   0
94  12  5   0
95  5   5   0
96  2   2   0
97  9   3   0
98  14  2   0
99  21  3   0
```

Bibliography

[1] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz, “Comparing elliptic curve cryptography and rsa on 8-bit cpus,” in *Cryptographic Hardware and Embedded Systems-CHES 2004*, pp. 119–132, Springer, 2004.

[2] A. Wander, N. Gura, H. Eberle, V. Gupta, and S. Shantz, “Energy analysis of public-key cryptography for wireless sensor networks,” in *IEEE Intl. Conf. on Pervasive Computing and Communications*, pp. 324–328, Mar. 2005.

[3] D. W. Carman, P. S. Kruus, and B. J. Matt, “Constraints and approaches for distributed sensor network security (final),” *DARPA Project report,(Cryptographic Technologies Group, Trusted Information System, NAI Labs)*, vol. 1, p. 1, 2000.

[4] J. P. Walters, Z. Liang, W. Shi, and V. Chaudhary, “Wireless sensor network security: A survey,” *Security in Distributed, Grid, Mobile, and Pervasive Computing*, vol. 1, p. 367, 2007.

[5] W. Diffie and M. E. Hellman, “New directions in cryptography,” *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[6] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[7] V. S. Miller, “Use of elliptic curves in cryptography,” in *Advances in Cryptology - CRYPTO85 Proceedings*, pp. 417–426, Springer, 1986.

[8] “Recommended elliptic curve domain parameters.” `www.secg.org/collateral/sec2.pdf`, 2000.

[9] P. C. Kocher, “The ssl protocol version 3.0,” tech. rep., Internet Draft, Netscape Communications Corporation, 03/04/96, 1996.

[10] Y. Wang, G. Attebury, and B. Ramamurthy, "A survey of security issues in wireless sensor networks," *Communications Surveys Tutorials, IEEE*, vol. 8, pp. 2–23, Second 2006.

[11] M. Brown, D. Cheung, D. Hankerson, J. L. Hernandez, M. Kirkup, and A. Menezes, "Pgp in constrained wireless devices," in *Proceedings of the 9th USENIX Security Symposium*, vol. 9, p. 19, 2000.

[12] M. H. Ahmed, S. W. Alam, N. Qureshi, and I. Baig, "Security for wsn based on elliptic curve cryptography," in *2011 International Conference on Computer Networks and Information Technology (ICCNIT)*, pp. 75–79, IEEE, 2011.

[13] J. Zutter, M. Thalmaier, M. Klein, and K.-O. Laux, "Acceleration of rsa cryptographic operations using fpga technology," in *20th International Workshop on Database and Expert Systems Application, 2009. DEXA'09*, pp. 20–25, IEEE, 2009.

[14] P. Ganesan, R. Venugopalan, P. Peddabachagari, A. Dean, F. Mueller, and M. Sichitiu, "Analyzing and modeling encryption overhead for sensor network nodes," in *Proc. of the 2nd ACM Intl. Conf. on Wireless Sensor Networks and Applications*, pp. 151–159, 2003.

[15] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography.* CRC press, 1996.

[16] R. L. Rivest, "The rc5 encryption algorithm," in *Fast Software Encryption*, pp. 86–96, Springer, 1995.

[17] D. Eastlake and P. Jones, *US secure hash algorithm 1 (SHA1).* RFC Editor, 2001.

[18] R. Rivest, *The MD5 message-digest algorithm.* RFC Editor, 1992.

[19] D. J. Wheeler and R. M. Needham, "Tea, a tiny encryption algorithm," in *Fast Software Encryption*, pp. 363–366, Springer, 1995.

[20] Y. W. Law, S. Etalle, and P. H. Hartel, *Assessing Security in Energy-Efficient Sensor Networks.* Springer, 2003.

[21] R. L. Rivest, M. Robshaw, R. Sidney, and Y. L. Yin, "The rc6tm block cipher," in *First Advanced Encryption Standard (AES) Conference*, 1998.

[22] J. Daemen and V. Rijmen, *AES proposal: Rijndael.* Citeseer, 1999.

[23] M. Matsui, "New block encryption algorithm misty," in *Fast Software Encryption*, pp. 54–68, Springer, 1997.

[24] ETSI/SAGE, "Specification of the 3gpp confidentiality and integrity algorithms document 2: Kasumi specification,," 3GPP, 1999.

[25] K. Aoki, T. Ichikawa, M. Kanda, M. Matsui, S. Moriai, J. Nakajima, and T. Tokita, "Camellia: A 128-bit block cipher suitable for multiple platforms-design and analysis," in *Selected Areas in Cryptography*, pp. 39–56, Springer, 2001.

[26] Y. W. Law, J. Doumen, and P. Hartel, "Benchmarking block ciphers for wireless sensor networks," in *2004 IEEE International Conference on Mobile Ad-hoc and Sensor Systems*, pp. 447–456, IEEE, 2004.

[27] L. Eschenauer and V. D. Gligor, "A key-management scheme for distributed sensor networks," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pp. 41–47, ACM, 2002.

[28] H. Chan, A. Perrig, and D. Song, "Random key predistribution schemes for sensor networks," in *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, pp. 197–213, IEEE Computer Society, 2003.

[29] D. Liu, P. Ning, and R. Li, "Establishing pairwise keys in distributed sensor networks," *ACM Transactions on Information and System Security (TISSEC)*, vol. 8, no. 1, pp. 41–77, 2005.

[30] R. Di Pietro, L. V. Mancini, and A. Mei, "Random key-assignment for secure wireless sensor networks," in *Proceedings of the 1st ACM workshop on Security of ad hoc and sensor networks*, pp. 62–71, ACM, 2003.

[31] W. Du, J. Deng, Y. S. Han, P. K. Varshney, J. Katz, and A. Khalili, "A pairwise key predistribution scheme for wireless sensor networks," *ACM Transactions on Information and System Security (TISSEC)*, vol. 8, no. 2, pp. 228–258, 2005.

[32] W. Du, J. Deng, Y. S. Han, S. Chen, and P. K. Varshney, "A key management scheme for wireless sensor networks using deployment knowledge," in *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, vol. 1, IEEE, 2004.

[33] D. Hwang, B.-C. Lai, and I. Verbauwhede, "Energy-memory-security tradeoffs in distributed sensor networks," in *Ad-Hoc, Mobile, and Wireless Networks*, vol. 3158 of *Lecture Notes in Computer Science*, pp. 70–81, Springer Berlin Heidelberg, 2004.

[34] S. Zhu, S. Setia, and S. Jajodia, "Leap+: Efficient security mechanisms for large-scale distributed sensor networks," *ACM Transactions on Sensor Networks (TOSN)*, vol. 2, no. 4, pp. 500–528, 2006.

[35] G. Zeng, X. Dong, and J. Bornemann, "Reconfigurable feedback shift register based stream cipher for wireless sensor networks," *IEEE Wireless Communications Letters*, vol. 2, pp. 559–562, Aug. 2013.

[36] D. Malan, M. Welsh, and M. Smith, "A public-key infrastructure for key distribution in tinyos based on elliptic curve cryptography," in *IEEE Communications Society Conf. on Sensor and Ad Hoc Communications and Networks*, pp. 71–80, Oct. 2004.

[37] M. Hell, T. Johansson, and W. Meier, "Grain: a stream cipher for constrained environments," in *International Journal of Wireless and Mobile Computing*, pp. 86–93, Jan. 2007.

[38] "eSTREAM: the ECRYPT Stream Cipher Project." `http://www.ecrypt.eu.org/stream/`.

[39] M. Hell, T. Johansson, A. Maximov, and W. Meier, "A stream cipher proposal: Grain-128," in *2006 IEEE International Symposium on Information Theory*, pp. 1614–1618, IEEE, 2006.

[40] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," in *SIGARCH Comput. Archit. News*, pp. 93–104, Nov. 2000.

[41] New Wave Instruments, “Tables of M-Sequence Feedback Taps.” `http://www.newwaveinstruments.com/resources/articles/m_sequence_linear_feedback_shift_register_lfsr.htm`.

[42] A. Wood and J. Stankovic, “Denial of service in sensor networks,” in *Computer*, pp. 54–62, Oct. 2002.

[43] Altera Corporation, *Quartus II Handbook Version 12.1.* Nov. 2012.

[44] Altera Corporation, *FPGA Power Management and Modeling Techniques.* Dec. 2010.

[45] D. Meintanis and I. Papaefstathiou, “Power consumption estimations vs measurements for fpga-based security cores,” in *Intl. Conf. on Reconfigurable Computing and FPGAs*, pp. 433–437, Dec. 2008.

[46] S. Fluhrer, I. Mantin, and A. Shamir, “Weaknesses in the key scheduling algorithm of rc4,” in *Selected Areas in Cryptography*, pp. 1–24, 2001.

[47] A. Biryukov and E. Kushilevitz, “Improved cryptanalysis of rc5,” in *Advances in Cryptology-EUROCRYPT’98*, pp. 85–99, 1998.

[48] I. Kuon and J. Rose, “Measuring the gap between fpgas and asics,” in *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, pp. 203–215, Feb. 2007.

[49] A. Gupta, J. Min, and I. Rhee, “Wifox: Scaling wifi performance for large audience environments,” in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, pp. 217–228, ACM, 2012.

[50] D. Grini, “RF Basics, RF for Non-RF Engineers.” `http://www.ti.com/lit/ml/slap127/slap127.pdf`, 2006.

[51] J. Zhao and R. Govindan, “Understanding packet delivery performance in dense wireless sensor networks,” in *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, pp. 1–13, ACM, 2003.

[52] W. W. Peterson and D. T. Brown, “Cyclic codes for error detection,” *Proceedings of the IRE*, vol. 49, no. 1, pp. 228–235, 1961.

[53] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," in *ACM SIGOPS Operating Systems Review*, vol. 34, pp. 93–104, ACM, 2000.

[54] C. Karlof, N. Sastry, and D. Wagner, "Tinysec: a link layer security architecture for wireless sensor networks," in *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, pp. 162–175, ACM, 2004.

[55] M. Luk, G. Mezzour, A. Perrig, and V. Gligor, "Minisec: a secure sensor network communication architecture," in *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, pp. 479–488, ACM, 2007.

[56] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway, "A concrete security treatment of symmetric encryption," pp. 394–403, IEEE Computer Society, 1997.

[57] M. Bellare, J. Kilian, and P. Rogaway, "The security of the cipher block chaining message authentication code," *Journal of Computer and System Sciences*, vol. 61, no. 3, pp. 362–399, 2000.

[58] P. Rogaway, M. Bellare, and J. Black, "Ocb: A block-cipher mode of operation for efficient authenticated encryption," *ACM Transactions on Information and System Security (TISSEC)*, vol. 6, no. 3, pp. 365–403, 2003.

[59] TinyOS Wiki, "Wireless Sensor Network Platform Hardware." `http://tinyos.stanford.edu/tinyos-wiki/index.php/Platform_Hardware`.

[60] T. Rappaport, *Wireless Communications: Principles and Practice*, vol. 2. Prentice Hall PTR New Jersey, 1996.

[61] S. Y. Seidel and T. S. Rappaport, "914 mhz path loss prediction models for indoor wireless communications in multifloored buildings," *IEEE Transactions on Antennas and Propagation*, vol. 40, no. 2, pp. 207–217, 1992.

[62] H. Nikookar and H. Hashemi, "Statistical modeling of signal amplitude fading of indoor radio propagation channels," in *2nd International Conference on Universal Personal Communications, 1993. Personal Communications: Gateway to the 21st Century. Conference Record*, vol. 1, pp. 84–88, IEEE, 1993.

[63] K. Sohrabi, B. Manriquez, and G. J. Pottie, "Near ground wideband channel measurement in 800-1000 mhz," in *1999 IEEE 49th Vehicular Technology Conference*, vol. 1, pp. 571–574, IEEE, 1999.