SECURITY FRAMEWORK FOR COMBINING CONFIDENTIALITY AND INTEGRITY

by

Jayagopal Narayanaswamy

Submitted in partial fulfillment of the
requirements for the degree of
Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
April 2015

I would like to dedicate this thesis to the force that drives me to succeed in every step I try to reach.

Table of Contents

List of Tables

List of Figures

Abstract

Radio Frequency Identification (RFID) technology represents objects uniquely in order to track their movement in the real world. To avoid an unauthorized entity tracking the object and to offer security, RFID systems require data encryption algorithms. However, they are severely resource-constrained and consequently, there has been an interest in the research community for proposing light-weight security protocols for RFID systems.

This thesis proposes two novel algorithms, namely, Standalone Simple Cryptographic Algorithm for RFID Systems (S-SCARS) that offers data security by combining integrity and authentication as part of the encryption process and Hybrid Symmetric Key Algorithm for Integrity Check, Dynamic Key Generation and Encryption (HIDE), that generates keys dynamically, along with integrity check parameters.

A software implementation of the proposed algorithms, and a hardware implementation using Xilinx have been completed in order to analyze the resource utilization of each algorithm. Furthermore, a security analysis of S-SCARS and HIDE has been evaluated using Cryptool in order to compare the proposed algorithms with standard algorithms such as AES, DES, RC4 and IDEA.

Acknowledgements

I would like to thank my supervisor, Prof. Srinivas Sampalli, for his support and encouragement in the successful completion of my research. I would like to remember and thank my mentors: Mr. Ravichandran D, Mr. Gopinath S, Mr. Vishwanath J and Prof. Srinivas Sampalli, who guided me in the right direction in choosing my career. I would also like to thank my parents Mr. Narayanaswamy Jayaraman and Mrs. Nirmala Narayanswamy for their constant encouragement and support.

List of Abbrieviations Used

AKE	Asymmetric Key Encryption
HIDE	Hybrid Symmetric Key Algorithm for Integrity Check, Dynamic Key Generation and Encryption
LFSR	Linear Feedback Shift Register
NLFSR	Non-Linear Feedback Shift Register
PRNG	Pseudo Random Number Generator
RBS	Redundant Bit Security
RFID	Radio Frequency Identification
S-SCARS	Standalone Simplified Cryptographic Algorithm for RFID Systems
SASI	Strong Authentication and Strong Integrity
SCARS	Simplified Cryptographic Algorithm for RFID Systems
SKE	Symmetric Key Encryption

Chapter 1

Introduction

1.1 Overview of Cryptography

In network security, there are different goals, such as authenticity, confidentiality, integrity, non-repudiation and digital signature, most of which can be achieved by encrypting the message using cryptographic algorithms [8]. Further, it can consist of different types, of which the most widely used are symmetric key algorithms, asymmetric key algorithms and hashing algorithms. A message is encrypted initially by a symmetric or an asymmetric key algorithm to provide confidentiality.

Symmetric key algorithms can be classified into two types: block ciphers and stream ciphers. Block ciphers split a message into blocks of identical sizes, which are encrypted with a pre-shared symmetric key block. A key for a block cipher remains the same or is derived from an initial key using functions such as the Feistel function [25]. In contrast, stream ciphers encrypt a stream of bits from a message with a key bit-by-bit using the logical exclusive-OR (XOR) operation. The key stream is generated by a PRNG through a fixed size input called the seed.

In stream ciphers, the encryption operation is simple but vulnerable to attacks such as distinguishing and key recovery attacks [16]. An elegant way for a stream cipher to generate a continuous stream of key bits is to use a Linear Feedback Shift Register (LFSR), which requires an n-bit seed value. A drawback with a LFSR is that an n-bit pattern may repeat in the key stream before completing all 2^n possible patterns [42]. To avoid this issue a Non-Linear Feedback Shift Register (NLFSR) has been proposed, but there are no generic designs for a NLFSR [12] [3]. In addition, in both block and stream ciphers, the key is derived from an initial key or a seed, therefore, knowledge of the initial key reveals information about the original message.

After the encryption process, an encrypted message is hashed to offer integrity. Hashing is a one-way mathematical function, which converts the variably sized messages into a fixed size output called a Message Digest (MD). The receiver verifies the

integrity of received message by hashing it and then compares it with the received MD.

In general, major applications uses asymmetric key encryption to provide authentication, symmetric key encryption to offer confidentiality and hashing for integrity. Traditionally, a key is used to encrypt a message using a symmetric key encryption algorithm to provide confidentiality, then the key is encrypted by an asymmetric key encryption algorithm to offer authentication, and finally, both are hashed to include integrity.

1.2 Motivation

To ensure major security goals, such as confidentiality, integrity and authentication, multiple cryptographic algorithms are used. Therefore, every message is subjected to multiple processes which make conventional cryptosystems computationally less efficient [26][7][6]. Recent trends in cryptographic algorithms propose combined integrated services, such as confidentiality with integrity or integrity with authentication, etc. This motivated the proposal of a the symmetric key encryption algorithm that can offer unified service to the system. On the one hand, a resource-constrained Radio Frequency Identification (RFID) system requires a simple data encryption algorithm that can prevent an unauthorized entity from tracking its object. On the other hand, hybrid cryptographic approaches are gaining popularity in the applications which require high security over data transmission with efficient computation. This thesis introduces two different data encryption schemes, the choice of which depends on the application requirement.

A RFID system, which offers both unique identification as well as automation, is an advancement over the traditional bar code system. As it is a cost effective technology, its usage is gaining popularity over many domains of inventory management [23]. Among other applications, the military domain needs more security which increases the necessity of implementing the cryptographic algorithms in RFID technology [30]. Since the RFID system is resource-constrained, using multiple algorithms to perform a data encryption is a challenging process. In addition, less computational power and a restricted number of gates allocated for security in RFID systems restrict them from performing heavy computational operations [4]. This has led to researchers

proposing lightweight and ultra-lightweight encryption algorithms such as Hummingbird [13], Redundant Bit Security algorithm (RBS) [20], Strong Authentication and Strong Integrity (SASI) [9], etc.

Recently, an algorithm called Hummingbird was proposed as a hybrid of the stream and block cipher approaches [13]. Hummingbird follows the traditional encryption process of block ciphers such as substitution and looping, while a key is derived through the stream cipher principle. However, this thesis is compared with Hummingbird as an example to illustrate the importance of the hybrid approach.

1.3 Proposed Approach

The purpose of this thesis is to offer a unified data encryption scheme depending on the application. A new approach for RFID systems called a Standalone Simplified Cryptographic Algorithm for RFID Systems (S-SCARS) is introduced for resource-constrained RFID systems. In addition, another cryptographic algorithm is proposed as a hybrid encryption scheme called a Hybrid Symmetric Key Algorithm for Integrity Check, Dynamic Key Generation and Encryption (HIDE) [31].

1. S-SCARS provides confidentiality, authentication and integrity check for resource-constrained devices such as RFID. It uses a simple exclusive-OR (XOR) operation, an expansion function, bit flipping and random number generation. Integrity check for the message is offered through the expansion function and bit flipping without using hashing. A tag and the server use random number generation to offer authentication, instead of using traditional asymmetric key encryption. Finally, two rounds of XOR operations provide confidentiality. Cryptool is used to perform security analysis and theoretical cryptanalysis is included as well.

2. A hybrid encryption algorithm, HIDE [31], requires a simple (Exclusive-OR) XOR operation. It uses the stream cipher approach to derive a key, while the block cipher approach is adopted in the encryption process. In our approach, a key stream is generated from a previous key block and an intermediate cipher text block, which is encrypted with a message block-by-block using the XOR operation. In addition to encryption, a fixed-size final key in each round is used as the MD that provides the integrity to the message. Since we use only the XOR operation throughout the

encryption process and provide the integrity check parameter without using any external hashing algorithm, we expect that our approach will reduce the computational complexity as well as increase performance. We have implemented the algorithm for proof of concept and we show that our algorithm can withstand such potential attacks as, differential, known cipher text, known plain text, distinguishing and key recovery attacks. We also conducted standard security analysis tests (such as, entropy analysis, periodicity check, frequency test, poker test, run test, serial test, etc.) and have included the results in the security analysis section.

1.4 Outline

This thesis is organized into the following sections. In Section 2, a literature review of this thesis is presented. In Section 3, the proposed algorithm for RFID systems will be discussed. In Section 4, the proposed algorithm for a hybrid approach will be discussed. In Section 5, the implementation of the proposed approaches are included. In Section 6, the security analysis of the proposed approaches are discussed. Section 7 concludes the paper.

Chapter 2

Background and Literature Review

2.1 Overview of Cryptography

In cryptography, encryption techniques can be broadly classified into two types: Symmetric Key Encryption (SKE) and Asymmetric Key Encryption (AKE). In general, SKE provides confidentiality and AKE offers authentication, where AKE has more computational overhead compared to SKE. A symmetric key algorithm can be further classified as block cipher and stream cipher. In both approaches the same key is used to encrypt as well as decrypt the message. A stream cipher encrypts the message bit-by-bit whereas a block cipher encrypts it block-by-block. Each block has a fixed number of bits (say 4, 8, 16, 32-bit block, etc.) [26][11].

Integrity is an important component of network security that prevents an anonymous entity from data manipulation. The integrity of the message can be achieved through hashing algorithms, such as, MD5, SHA1, SHA2, SHA3, etc. Traditional hashing is a mathematical one-way function which encrypts the message, but decryption is not possible. It converts any variable size message to a fixed size output called the Message Digest (MD). A collision between two MD is possible after hashing 2^n messages. However, as the encryption (or decryption) and hashing are disjointed operations, the use of hashing will lead to computational overhead on the system [26] [18] [27].

In recent years, a hybrid approach called Authenticated Encryption (AE) has provided integrity as well as authenticity for short messages [21]. However, it requires a Message Authentication Code (MAC), which is derived from the hashing technique. This creates further computational overhead. In addition, AE is suitable for both symmetric key and asymmetric key encryption mechanisms.

Signcryption was proposed for the asymmetric key algorithm [43][40], which replaced a traditional encrypt and sign practice by integrating message signatures (similar to MD) as a part of an encryption process. Zheng et al.'s [43] work on the

signcryption scheme is based on the theory that a combined computational cost of a signature (using hashing) and encryption will be less than their individual costs. In signcryption, part of the key is generated using a hashing algorithm to provide the integrity, and the same key is used to encrypt (using an asymmetric key algorithm) the message to offer confidentiality, integrity and authentication [43].

2.2 Scheme 1: S-SCARS

The feasibility of Elliptic Curve Cryptography, which is an asymmetric key algorithm for RFID systems, was investigated by Batina et al. [4]. They determined that the public key encryption mechanism of Schnorr's identification protocol was less expensive than Okamoto's technique for RFID systems. However, as the RFID is a resource-constrained system, they also concluded that Symmetric key encryption algorithms are preferable for RFID systems as they require less computation power.

Syamsuddin et al. [38] surveyed the Hash-Chain methods for RFID systems against vulnerable attacks (such as man in the middle, desynchronization, spoofing, replay attack, forward secrecy, etc.). Their work proves that existing protocols fail to provide integrated services such as confidentiality, integrity and authentication combined.

Jeddi et al. [20] proposed a symmetric key encryption algorithm called RBS (Redundant Bit Security) to offer confidentiality with integrity for RFID systems. They used the MAC algorithm to generate redundant bits, which helps offer integrity to the message. The redundant bits generated are inserted into the original message while encrypting. The encryption is performed with a smaller number of cycles as compared to traditional encryption schemes and that leads to less computational complexity. Later, the same approach was extended to offer authentication to the message. [21].

Narayanaswamy et al. [32] framed a symmetric key-based encryption algorithm for RFID systems, called Simplified Cryptographic Algorithm for RFID Systems (SCARS), which also offers confidentiality along with integrity without using external hashing algorithms. SCARS works on a framework of two rounds of XOR operations, an expansion and a bit flipping function. There are two 64-bit keys, namely, K_p and K_f, in which the key K_f is derived directly from K_p through the bit flipping function.

For an initial round of the XOR operation a 64-bit message is XORed with the key K_p followed by the expansion function which doubles its size. A 128-bit output will be processed through bit flipping, which is then XORed with the 128-bit key that was concatenated from the keys K_p and K_f. This approach offers a unified service of confidentiality and integrity, with reduced logic gates.

Hung-Yu Chien [9] proposed an ultra-lightweight authentication protocol for RFID systems, called Strong Authentication and Strong Integrity (SASI), using a simple bit-wise operation. It requires a static ID, two keys ($K1$ and $K2$) and pre-shared IDS between the server and tag. For a request from the reader, the tag responds with IDS or previous IDS. After successful authentication, the reader will generate a random value, and using previously shared keys, it will also generate A, B and C. The reader will share A, B and C, where the tag will extract A, B, C, n1 and n2, using the pre-shared ID, K1 and K2. The tag will generate D and share it with the reader, where authentication is achieved eventually. Though there are updated versions and drawbacks in SCSI, this algorithm is used as an example of a unified approach for offering authentication and integrity in a single approach, which is also an ultra light weight algorithm.

In summary, existing algorithms offer a single security goal independently or more security goals when integrated with other algorithms (e.g. RBS uses MAC [20]). S-SCARS provides confidentiality, integrity and authentication to the message independently, without support from any other algorithms.

2.3 Scheme 2: HIDE

In stream ciphers, an LFSR plays an important role in the generation of a key stream. One possible implementation of a Pseudo Random Number Generator (PRNG) is using an LFSR. An LFSR can be constructed by the consecutive assembly of shift registers. An n-bit LFSR produces a continuous bit stream from an n-bit seed. One of the drawbacks of an LFSR is that it will form a cycle on or before reaching 2^n patterns for a given n-bit seed (i.e. the same sequence of bits will be generated on or before reaching 2^n patterns) [42].

To overcome this drawback, an NLFSR is used [12] [3]. NLFSRs avoid the linearity problem and extend the cyclic period. A common way of constructing an NLFSR

is to use more than one LFSR connected through logic gates. Logic gates are used to choose the current LFSR output from a list of connected LFSRs. Thus, linearity can be broken easily and the cyclic period will be extended. Even though NLFSR acts as an alternative to LFSR, there is no NLFSR, which guarantees a long cyclic period. Furthermore, a general technique for constructing NLFSRs has been an open problem [28] [35] [17] [29].

Hybrid approaches that combine the benefits of both block and stream ciphers are the focus of recent research. Trivium [10] is the stream cipher that uses the block cipher principle for the key generation process, which is reviewed to present an example of a hybrid approach.

Engels et al. [13] proposed an SKE algorithm called Hummingbird that encrypts the message block-by-block and uses internal registers to update the key stream. It has a small block size of 64-bits, which is further divided into four smaller blocks of 16-bits each that are equivalent to the number of bits in the internal state register. Initially, the internal state register is loaded with random bits. Furthermore, the LFSR and the cipher text are used to regenerate the internal state register bits. It uses a 256-bit key that is split into four 64-bit keys to encrypt the message. Two more secret keys, namely, $K5$ and $K6$, are introduced and these are derived from the initial four keys. The Hummingbird algorithm has four rounds of substitution and permutation and uses four keys on each round to encrypt the message. In the fifth round, the output is encrypted with key $K5$ followed by a final substitution and then encryption with key $K6$. This approach is more suitable for resource-constrained devices such as RFID systems. It is resistant to differential as well as linear attacks [13]. However, this algorithm is also an example of a hybrid approach and HIDE does not inherit any properties used in Hummingbird.

Kutuboddin and Krupa [22] have extended the research on the Hummingbird algorithm to increase its computational efficiency. They propose an approach, which replaces the traditional Hummingbird modulo operation by introducing a Field Programmable Gate Array (FPGA). As the proposed approach uses the XOR operation, computational efficiency is better compared to the original version.

In conclusion, existing algorithms are weak when it comes to generating a key or the encryption process leads to computational overhead. To fill a gap between

computational overhead and to support a strong dynamic key generation process to encrypt the message, HIDE uses a simple XOR operation for better computation, and Intermediate Cipher Text (ICT) to generate the key dynamically.

Chapter 3

Proposed Approach 1: S-SCARS

3.1 Overview

RFID is a resource-constrained technology, which contains three entities, namely, server, reader and tag. An RFID reader queries the tag to obtain the unique identification number stored in it. The reader will forward the response from the tag to the server (where the data is processed). The tag has lack of storage memory, which restricts the performance of a cryptographic operation on the message (tag's unique identification number) that leads to data insecurity. As discussed in the Literature Review, symmetric key encryption is more suitable for RFID systems and the proposed algorithm uses it in three parts, namely — an encryption part, a key generation part and a random number generation part (used for authentication purpose). The encryption part uses a basic XOR operation, an expansion function and a bit flipping function, which combine to offer confidentiality and integrity. A random number generation operation is included at both the tag and server side to provide authentication. Finally, a bit flipping function is used as well for the key generation part. Our approach offers confidentiality, integrity and authentication to the message without using an external algorithm (such as hashing or asymmetric key encryption mechanism), which reduces the logic gate usage that makes this approach more suitable for RFID technology.

3.2 Architecture

With the proposed approach, a typical communication proceeds as follows — the server first queries the tag along with a piece of information that is used for authentication, which is derived from pre-shared nonce bits. A key to encrypt the message is pre-shared between the server and the tag, which leads to the encryption process based on key synchronization. The tag verifies the received information to ensure

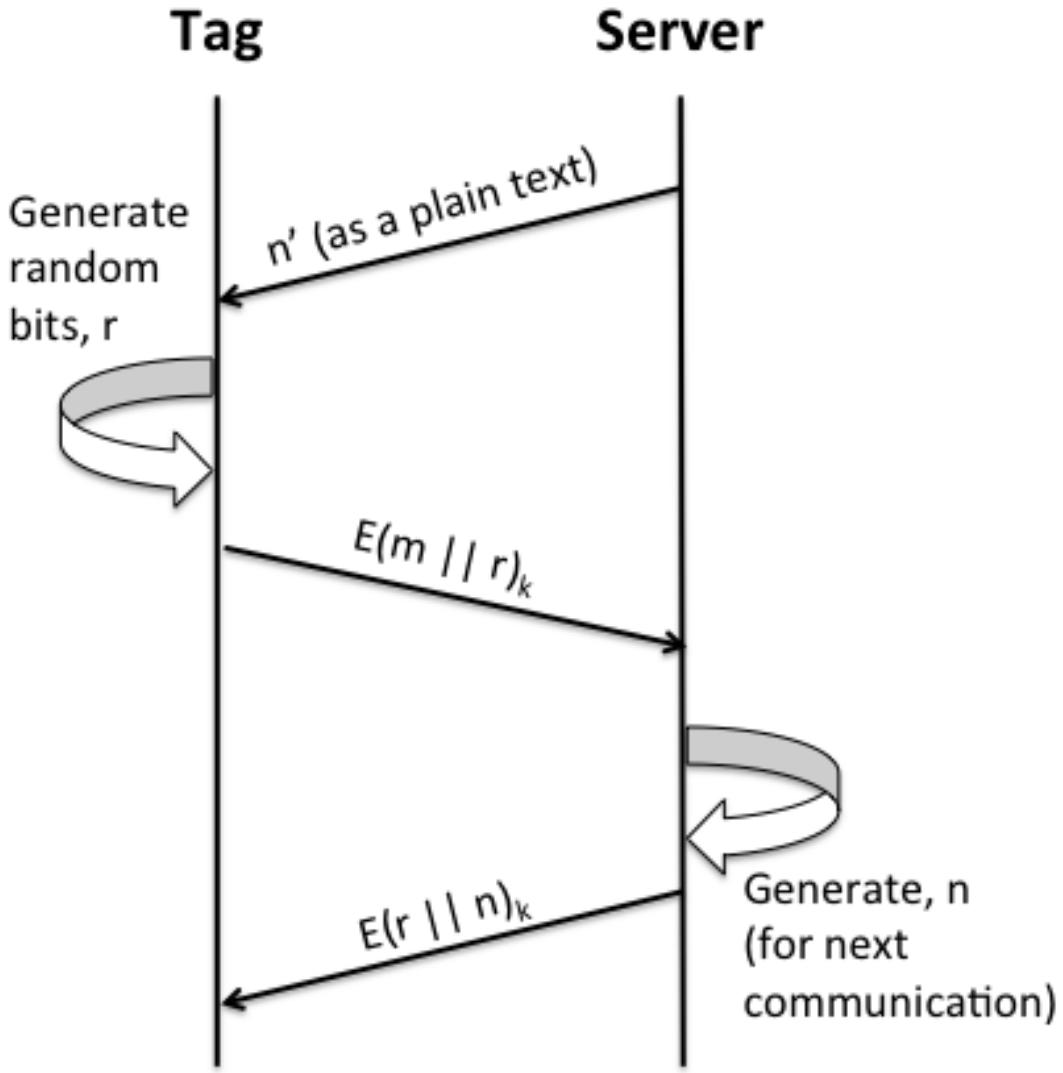


Figure 3.1: The architecture of the proposed approach

authentic communication and then generates random bits that are used for the later authentication process. The message and newly generated random bits are encrypted by the tag, which is forwarded to the server. The server decrypts it to retrieve the message and generates random bits that will be used as nonce for the next communication. Further, the server encrypts the newly generated nonce and the previously received random bits, which are forwarded to the tag. Finally, the tag compares the received random bits with the existing ones and then updates the nonce value, if the random bits are same. The architecture of this approach is shown in Figure 5.1.

3.3 Assumptions

It is assumed that the pre-shared key and nonce are securely transmitted between the server and tag. Further, it is assumed that the communication procedure between the server and the reader is well enough known that the present work deals only with communication between the server and the tag.

3.4 Notations

Table 3.1 contains a list of the notations used in this work.

Table 3.1: Notations for S-SCARS

MSB	Most Significant Bit
LSB	Least Significant Bit
E	Encryption
D	Decryption
m	Message
r	Random bits
n	Nonce
$n^{'}$	Authentication code
l	Message length (or) LSB of message
$Exp()$	Expansion function
$Cmp()$	Compression function
$BF()$	Flipping with respect to encryption or decryption
$XOR()$	Logical XOR Operation
i	Represents current message bit position
j	Calculates the bit to be flipped
$\|\|$	Concatenation operation
x	Represents current flipping bit
K_p	Key for first round of current message
K_f	Key for first or second round of next or current message respectively

3.5 Design

An encryption mechanism ensures that the message is transmitted secretly between legitimate users. Symmetric or asymmetric key encryption is used to encrypt the message, which provides confidentiality and prevents external parties from accessing the data. To avoid data manipulation by an unauthorized entity, a hashing algorithm is used to offer integrity to the message. Authentication ensures that only legitimate users access the data, which can be achieved through an asymmetric key encryption mechanism. This approach proposes the use of a less computationally intensive symmetric key encryption algorithm for resource-constrained RFID systems which accomplishes the important security goals (confidentiality, integrity and authentication) without using external algorithms.

3.5.1 Key Generation

The algorithm uses two keys — a present key K_p and a future key K_f. Both keys K_p and K_f are $(64 + 16)$-bits long (which includes a 16-bit nonce n). The present key K_p and nonce n were pre-shared, where K_f is derived directly from the concatenated K_p and n values through the bit flipping function. After the bit flipping, the first 64-bits are used as future key K_f and the last 16-bits are used as authentication bits $n^{'}$, which is used to authenticate the server at each communication of an initial query to the tag from the server. The nonce n is generated by the server for every successful communication, updating it at both the server and tag ends. Finally, the key K_f will be used as K_p and vice-versa. The first 64-bits from the key K_p is XORed with the message for the first round. For the last round of encryption, the first 64-bits from the keys K_p and K_f are combined to form a 128-bit key, which is XORed with the 128-bit output from the bit flipping function that will be the ciphertext.

3.5.2 Initial Communication

An initial message from the server to the tag has the authentication information for the current communication, which varies from one communication to another. The authentication code, $n^{'}$, is generated and stored on both the server and the tag before the current communication is invoked. The server initiates the process of

communicating with the tag by sending a 16-bit authentication code, n', as plain text, which was generated and pre-shared between them in the previous communication. The tag verifies the authentication code and proceeds with the encryption process.

3.5.3 Encryption

The proposed approach uses an expansion function that is changed from a traditional one, as well as a new operation called *bit flipping*. The expansion function uses a logical exclusive-OR gate to offer integrity to the message, which doubles the original message size. Bit flipping ensures that the expansion function offers integrity and

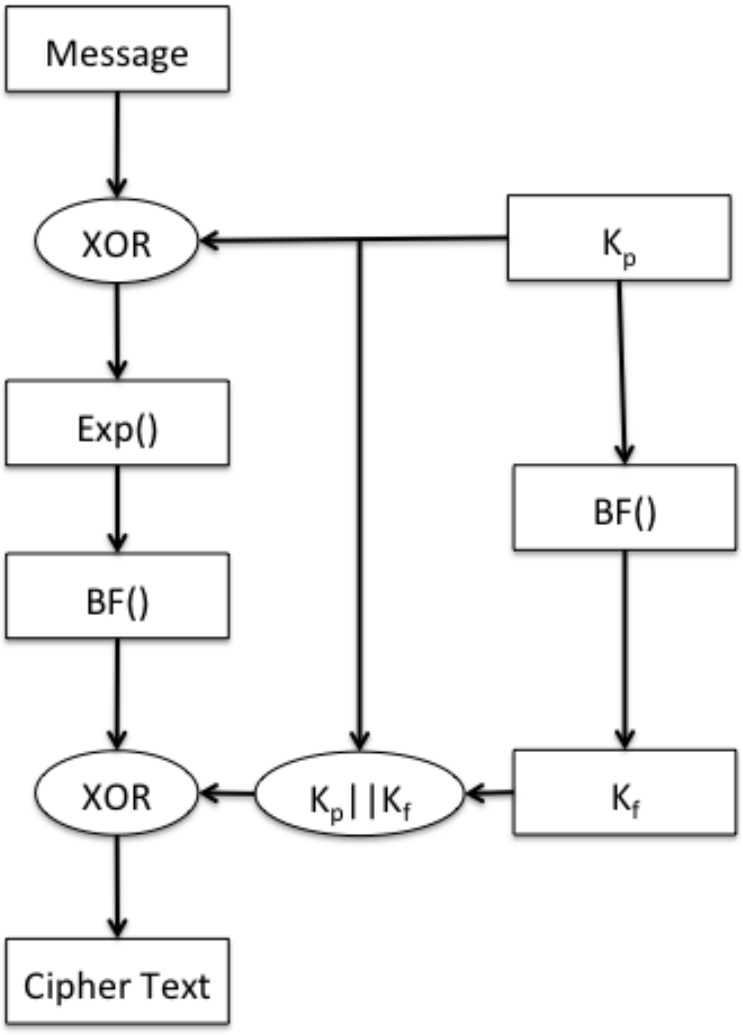


Figure 3.2: Overview of the encryption process — S-SCARS

is a supplementary process of the approach. Further explanations of the expansion and bit flipping functions will appear in later sub-sections. Finally, two rounds of the XOR operation provides confidentiality to the message.

In this protocol for RFID systems a 48-bit message is concatenated with 16-bits that is generated randomly (say, random bit r) by the tag, which is XORed with the first 64-bits of the present key, K_p. This is the first round of the XOR operation, which will be followed by the expansion and bit flipping operations, which will be discussed in the next section. Data after the expansion and bit flipping will again be XORed for the second round with the first 64-bit future key, K_f, which will be the final cipher text. Figure 5.2 illustrates the encryption process.

Expansion function, $Exp(m)$

The expansion function introduces additional bits to the original message, m, which will generate a pattern to offer an integrity parameter to it. A logical XOR operation is used to perform the expansion of the message that is concatenated with the random bits. After an expansion operation the 64-bit output from the first round of the XOR operation will be expanded to 128-bit (i.e. every 4-bits in the message starting from the MSB is converted to 8-bit output after an expansion function), which converts a 64-bit input message into a 128-bit output. This function will take the message, m, as an input, and considers evey four bits (starting with the most significant bit (MSB), m_0, until the least significant bit (LSB), m_{l-1}, where l represents the length of the message in bits) to compute the integrity parameter. The expansion function, $Exp(m)$, is defined as follows:

$$Exp(m_i, m_{i+1}, m_{i+2}, m_{i+3}) = m_i||m'_1||m_{i+3}||m_{i+1}||m_{i+2}||m'_2||m_{i+1}||m_{i+3} \quad (3.1)$$

Where, $m'_1 = XOR(m_i, m_{i+1})$;
$m'_2 = XOR(m_{i+2}, m_{i+3})$;
m_i is the i^{th} bit of the message m;
$XOR(m_i, m_{i+1})$ represents the exclusive-OR operation on the ${m_i}^{th}, {m_{(i+1)}}^{th}$ bit positions;
$Exp()$ represents the expansion function for encryption;
|| represents the concatenation operation.

The expanded message is proceeded by the bit flipping function, as discussed in detail in the next section.

Bit flipping function for encryption, $BF_{enc}(m)$

The bit flipping function (along with the expansion operation) provides an extra layer of data integrity. The proposed protocol applies the bit flipping function on the 128-bit output of the expansion function, where a bit is flipped based on 2-bit input (beginning with the MSB, m_0, until the least-significant bit, $m_{(2l-1)}$). For a given 128-bits, 64-bits are flipped, which will be followed by the final XOR operation with the concatenated keys, K_p and K_f. The expansion function, $BF_{enc}(m)$, is defined as follows:

$$m_x = \neg m_x \tag{3.2}$$

Where, $\neg$ represents bit inversion or bit flipping;
$x = (i + j + 2)\%128$;
$\forall m_i \in m, i = [0...127], j = [0...3]$;
i and $i + 1$ were the indication of bit positions;
j indicates the flipping bit with respect to the indication bit i;
m_x represents the bit to flip in a message m.

The values i and j are used to find the flipped bit that is represented by m_x. A variable i indicates the bit position in a message and j would be calculated by the bit combination of i and $i + 1$. Any bit that is flipped in a block is reflected only within that block (i.e. the bit operations performed in a block are cyclic).

Finally, the tag responds to the server with the encrypted message that contains the authentication information. The server decrypts the received message. The decryption procedure is discussed in detail in the next section.

3.5.4 Decryption

The decryption process in this approach is an inverse of encryption. Initially, the 128-bit ciphertext is XORed with a 128-bit key that is concatenated from the keys K_p and K_f, which is followed by the bit flipping function and compression operation that results in restoring the message size to 64-bits from 128-bits. Finally, the 64-bit

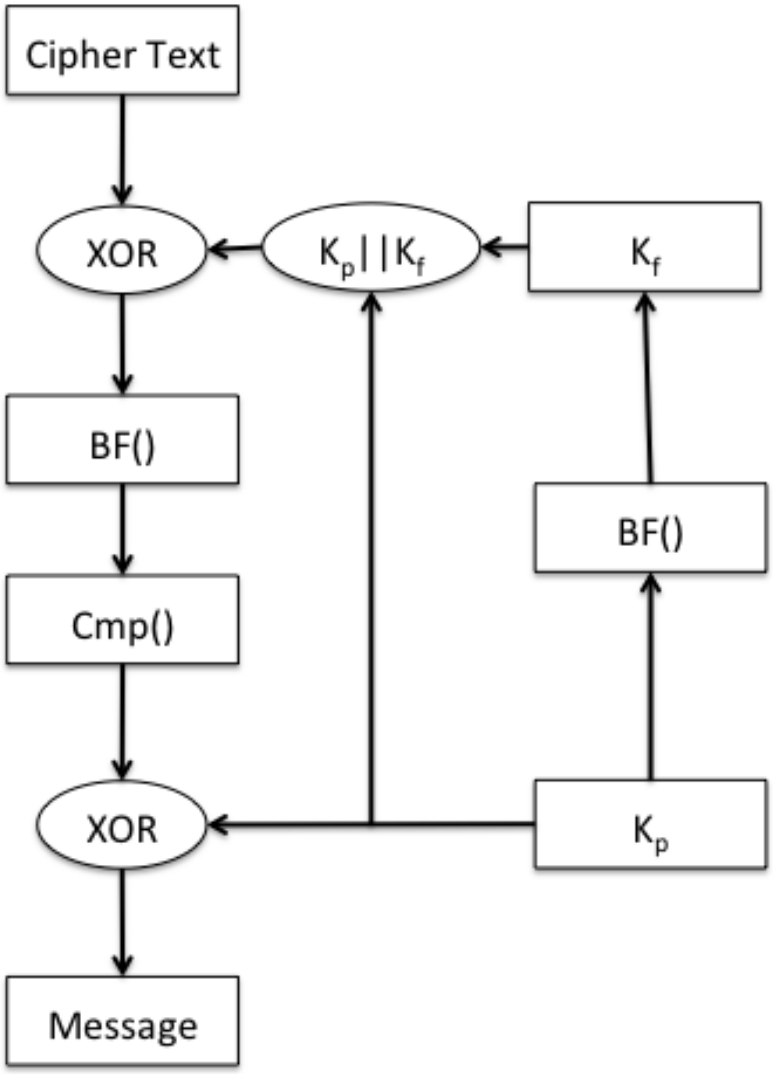


Figure 3.3: Overview of the decryption process — S-SCARS

output is XORed with the key, K_p, which yields the plain text that contains the actual message, M, with the random bits, R, (authentication information). The key generation process for decryption is the same as the encryption process. Figure 3.3 illustrates the decryption process.

Bit flipping function for decryption, $BF_{dec}(m)$

In this section, the proposed protocol performs the following bit flipping operation in a manner opposite to the encryption, i.e. beginning with the LSB, m_{2l-1}, until the MSB, m_0. The bit flipping function, $BF_{dec}(m)$ consumes 128-bits and generates the

same number of bits as the output from the initial XOR operation, which is similar to encryption.

$$m_x = \neg m_x \tag{3.3}$$

Where, $\neg$ represents bit inversion or bit flipping;
$x = (i + j + 2)\%128$;
$\forall m_i \in m, i = [127...0], j = [0...3]$;
i is the indication of bit positions;
j indicates the flipping bit with respect to the indication bit i;
m_x represents the bit to flip in a message m.

Compression function, $Cmp(m')$

For each pair of bits of the ciphertext (m'), beginning with the MSB, m'_0, until the LSB, m'_{2l-1} [$2l$ represents the length of the expanded ciphertext], the compression function, $Cmp(m')$, is derived as follows. If $XOR(m_i, m_{i+3})$ is equal to m_{i+1}, m_{i+3} is equal to m_{i+7}, $XOR(m_{i+4}, m_{i+7})$ is equal to m_{i+5}, and if m_{i+6} is equal to m_{i+3}, then, the received cipher text is assumed not to have been modified (i.e. the message integrity is verified) and the decrypted message bits will be $m_i, m_{i+3}, m_{i+4}, m_{i+7}$.

Finally, the 64-bits from the compression operation is XORed with the present key, K_p that will yield the plain text, which is composed of the actual message m and the random bits r.

3.5.5 Updating Nonce N

The server generate 16-bits randomly, which will act as the nonce, n, for the next communication. The 16-bit nonce generated by the server will be concatenated with the random b its, r, which was received from the tag. The first half of the concatenated message from the server will be composed of the random bits, r, and the second half will be the nonce, n. On the one hand, the concatenated message will be encrypted by the server using the same keys that were used for decryption (the encryption process was already discussed earlier), which will be transmitted to the tag to update the key. To encrypt the concatenated message, the 32-bit message is XORed with the first 32-bits of the present key, K_p, followed by the expansion function on the 32-bit

output and then the bit flipping operation. Finally, the 64-bit output from the bit flipping will be XORed with the future key, K_f. On the other hand, the server will update the key by replacing the $n^{'}$ in the key, K_f with the new nonce, n, which will act as K_p, for the next communication.

The tag will decrypt the received message to obtain the tag-generated random bits r and the server-generated nonce n. If the tag-generated (and stored) random bits are the same as the received one then the nonce n will be updated with the key (similar to the server key update procedure).

3.6 Example

This section explains the proposed approach with an example (Figure 3.4 and Figure3.5). The message size is 8-bits (the unique identification number of the tag) and the keys (K_p and K_f) are 8-bits each. In addition, the random bits, r, the nonce, n and the authentication code, $n^{'}$, each have 2-bits.

A pre-shared present key, 10001011, is concatenated with the nonce, 01, (the random bits generated by the server as shared with the tag in the previous communication) to generate a future key, 11011011, and an authentication code, 11, through the bit flipping function. The server establishes communication with the tag by sending the authentication code (11). After the tag verifies the authentication code (11), it generates a random bit (10), which is concatenated with the message (001110). A concatenated value, 00111010, is XORed with the present key (10001011), which result (10110001) will be expanded to 16-bit (1110100100100101). An expanded value will undergo bit flipping that is 1011111100001101 will be XORed with the future key, 11011011, that results in the cipher text (0011010011010110). The tag will forward the encrypted message to the server, which will decrypts the received cipher text to obtain the actual message and a random bit generated by the tag.

Now, the server will generate a random bit (10) that can be used as a nonce for the next communication. A nonce (10) generated by the server is concatenated with the received random bit (10) that will be XORed with the first 4-bits of the present key (1000), which results in 0010. The output will be expanded and bit flipped then XORed with the future key (11011011) to obtain a cipher text (10010101). The server will forward the encrypted message to the tag that will decrypt the received cipher

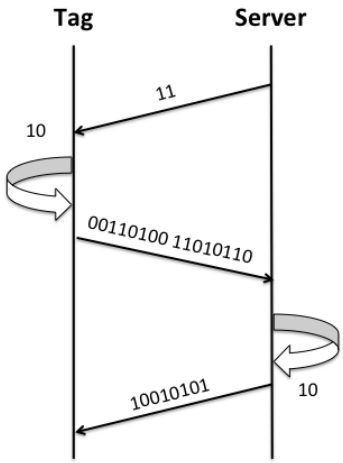


Figure 3.4: An example of the proposed approach — Data Communication

text and compares the random bit (10) with the existing one. If both are same then the tag will accept the nonce from the server and updates the key correspondingly.

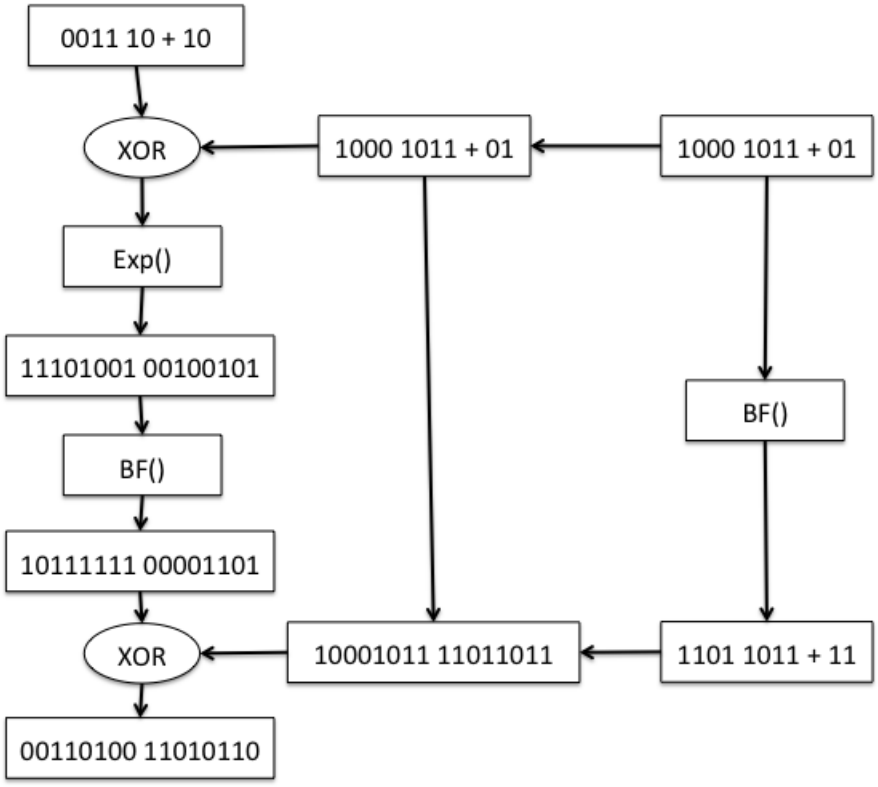


Figure 3.5: An example of the proposed approach — Encryption

Chapter 4

Proposed Approach 2: HIDE

4.1 Overview

In block ciphers, common modes of encryption such as Cipher Block Chaining (CBC), Propagating Cipher Block Chaining (PCBC) and Cipher Feedback (CFB) use the cipher text as part of the encryption. This may lead to some information about the original message being revealed to an attacker. To avoid this issue, this approach uses an Intermediate Cipher Text (ICT) to generate a dynamic key. Since the dynamic key on each block of the message encryption, the cipher text bits are completely random.

A hybrid encryption algorithm is proposed, which includes both stream cipher and block cipher features. This approach consist of two parts: a key generation part (based on stream ciphers) and two rounds of encryption (or decryption) using a basic XOR operation (employing message blocks as in block ciphers). The message is encrypted block-by-block with different (block) keys is used on each message block. The size of one block in the message is 128-bits in sixteen 8-bit chunks.

Figure 4.1 illustrates an overview of the block structure and Figure 4.2 illustrates the overall encryption process. This approach requires two initial keys that encrypt the first block in each round. Except for the initial key, every successive key (to encrypt each block) is derived from the bits in the current key block and an ICT block. In general, blocks of message are encrypted using the blocks of keys generated using the stream cipher scheme. Key generation and encryption follows a chained approach, with the current (block) keys and ICTs used to generate (block) keys for the encryption of the subsequent message blocks. The encryption and key generation processes are described later in this section.

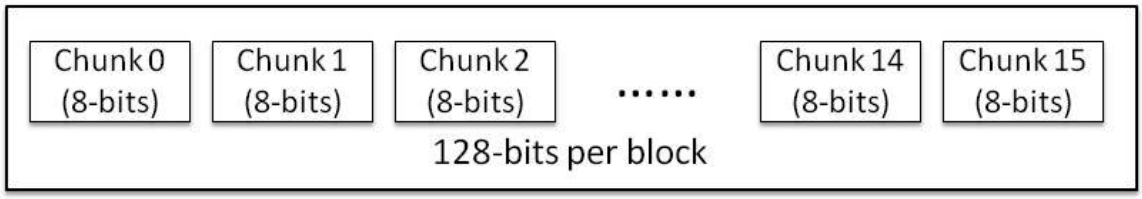


Figure 4.1: A Block Structure

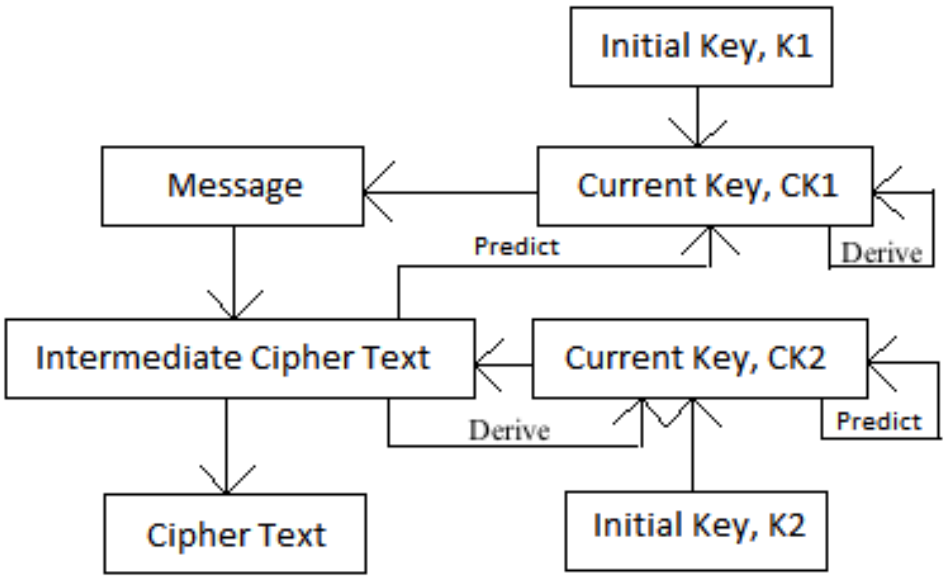


Figure 4.2: Overview of the encryption process — HIDE

4.2 Assumptions

It is assume that the initial key used to encrypt the first block of the message in each round is generated by the sender and securely transmitted to the receiver. The key used to encrypt the message is randomly chosen from a set of strong keys.

4.3 Notation

Table 4.1 contains a list of notations that are used in the proposed approach.

Table 4.1: Notations for HIDE

K_a	Range of keys to encrypt the message in Round One
K_b	Range of keys to encrypt the message in Round Two
n	Last block
l	Current block
i_x	Prediction bit
M	Message to be encrypted
$m_1, m_2 ... m_l$	Message blocks
XOR	Logical bit-by-bit XOR operation
$\delta()$	Derivation function
$\rho()$	Prediction function
$\gamma()$	Inversion function
M_x	Message block x
K_{ax}	Round One key block x
K_{bx}	Round Two key block x
ICT_x	Intermediate Cipher Text block x
CT_x	Cipher Text block x
$C_{\alpha,\beta}$	Chunk β of component α (α can be M_x, ICT_x, K_{ax}, K_{bx}, CT_x)
KG1	Key generation for Round One
KG2	Key generation for Round Two

4.4 Design

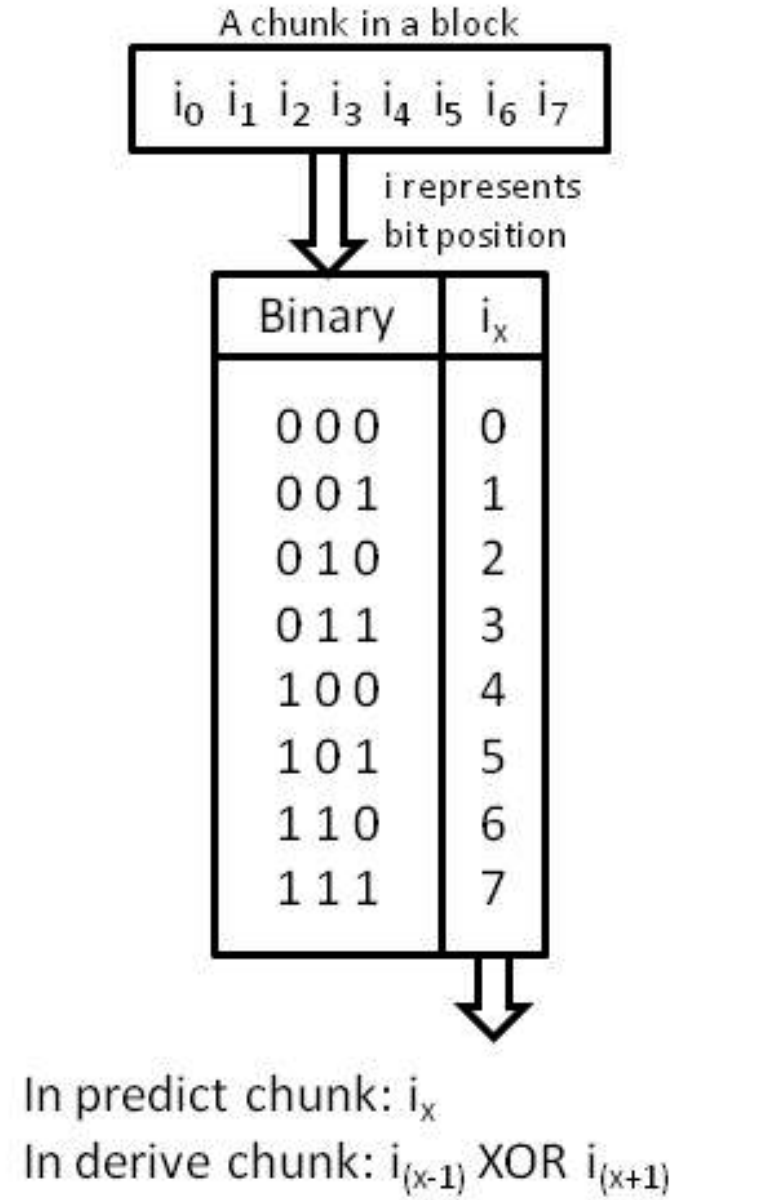
4.4.1 Key Generation

Since it is assumed that the key is known only to the sender and the receiver, we generate successive keys from the previous keys with the help of an intermediate cipher text (ICT). The term "intermediate cipher text" represents an encrypted message after Round 1, whereas the actual cipher text is generated after Round 2 (as will be discussed in detail in the encryption section). The key generation process in this approach is different for both rounds. Each round uses a different initial key (K_{a0} & K_{b0}) to encrypt the first block in the message. To encrypt each successive block in the message, we introduce a combination of *prediction* and *derivation* techniques to generate successive keys ($K_{a1...n}$ for Round 1 and $K_{b1...n}$ for Round 2) from a previous key block and an ICT block.

In *prediction*, three bits in every chunk per block are used to choose the i_x^{th} bit within the respective chunk. The value of i_x is based on the three binary bits chosen in the chunk. The value of i_x is a decimal representation of the 3-bit binary. In a chunk, 3-bits are selected in a clockwise direction starting from the most significant bit (MSB) in the chunk to the least significant bit (LSB) to choose one out of eight possible values for i_x, which is repeated 8 times starting from the MSB to the LSB. The value of i represents the position of the binary bit in the chunk. Each value of i_x predicts a new bit that is used to generate chunks per block in every successive key. As shown in Figure 4.3, three bits are rotated in a clockwise direction to yield one out of eight possible values for i_x.

In *derivation*, the successive key is generated by XORing the $(i_x - 1)^{th}$ position of the binary bit with the $(i_x + 1)^{th}$ position of the binary bit to form a new bit. The same process is repeated in every chunk per block to generate a whole key for encrypting the successive block in the message. The whole process of key generation offers a randomness in the key stream without forming a cycle.

The process for applying *prediction* and *derivation* in combination to generate a key is as follows. In Round 1, the key is derived from key K_a series based on the ICT block's prediction. In contrast, Round 2's successive keys are derived from the ICT block based on the key block's prediction. To generate a key, the *prediction* and



	Predict i_x	Derive $i_{(x-1)}$ XOR $i_{(x+1)}$
KG1	ICT	Key
KG2	Key	ICT

Figure 4.3: Key Generation Process Overview

derivation combination is applied alternatively on an ICT block and a key block over two rounds.

4.4.2 Encryption

Initially, in Round 1, four out of eight bits (say $0, 2, 3, 5$) in every chunk in a key K_{a0} are inverted. The first chunk in the initial key K_{a0} is XORed with the first chunk in the first block of the message, which produces the first chunk in ICT_0 (which represents the ICT first block). Again, four out of eight bit (say $2, 4, 5, 7$) in every chunk in ICT_0 are inverted (this process is common throughout the ICT blocks), which is followed by successive key generation operations (already discussed in detail earlier). The output of the key generation operation will be the first chunk in the key K_{a1} to encrypt the first chunk of the next message block. The first chunk of key K_{a1} is XORed with the second chunk of key K_{a0} that will be used to encrypt the second chunk in the first block of the message. In conclusion, every first chunk in each block of the message is directly XORed with the first chunk of the respective key, whereas the successive chunks in each block in the message are encrypted with the XORed output of the successive chunk in the present key and the currently derived chunk for the next key. For example, the 4^{th} chunk in message $C_{M_2,4}$ will be encrypted with the XORed output of the 4^{th} chunk in the current key $C_{K_{a2},4}$ and the 3^{rd} chunk in the next key $C_{K_{a3},3}$.

Round 2 follows the same procedure as Round 1, except that the message and initial key K_{a0} will be replaced by the ICT and key K_{b0} respectively. In addition, the inversion operation is applied on bits $1, 4, 6, 7$ and $0, 1, 3, 6$ for the key and the ICT respectively. The output of Round 2 is the final cipher text. Every block in a message is XORed with the key block to generate a cipher text. Figure 4.4 illustrates the encryption process.

4.4.3 Decryption

Figure 4.5 illustrates the decryption process, which is symmetric with encryption.

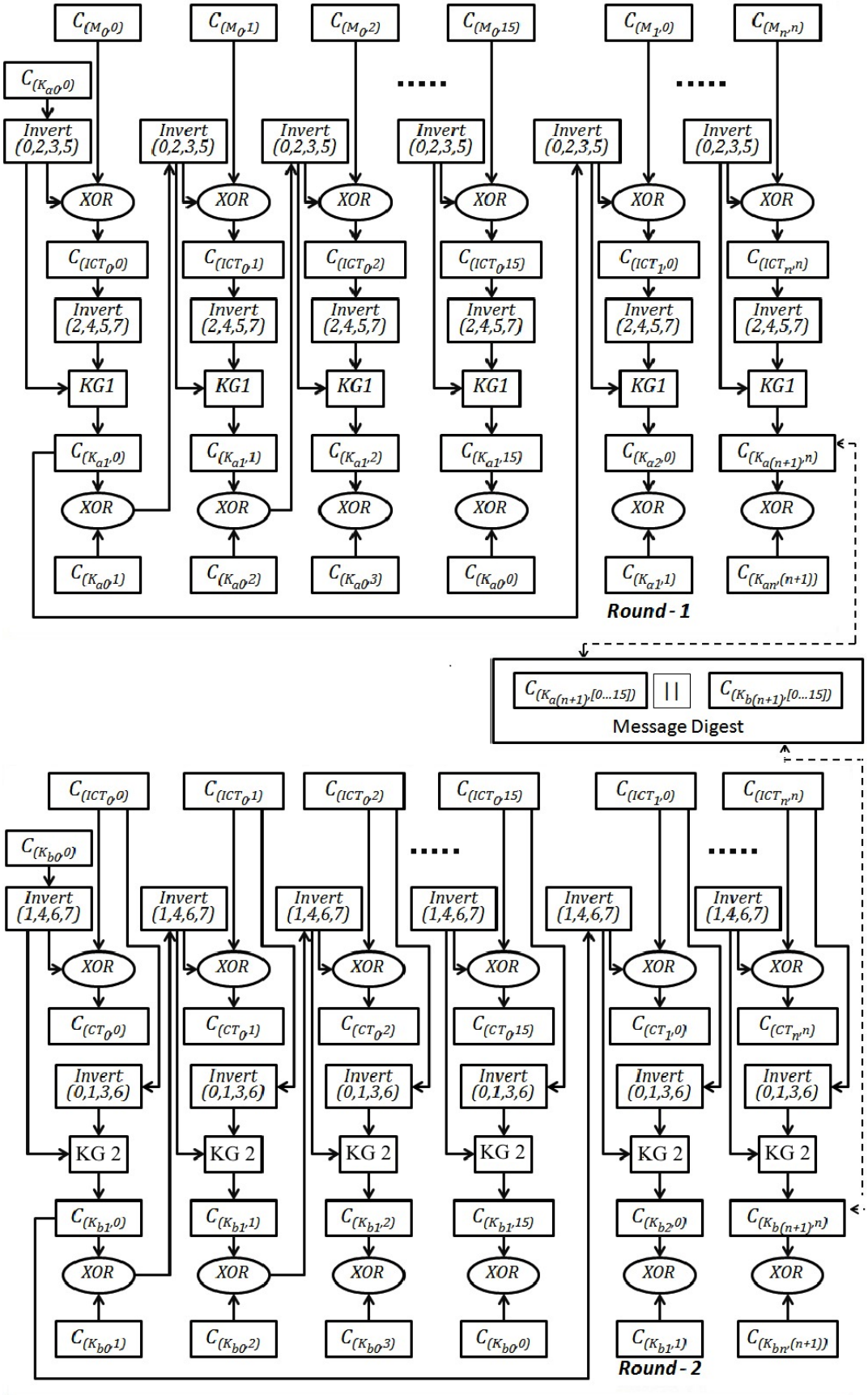


Figure 4.4: Encryption Process

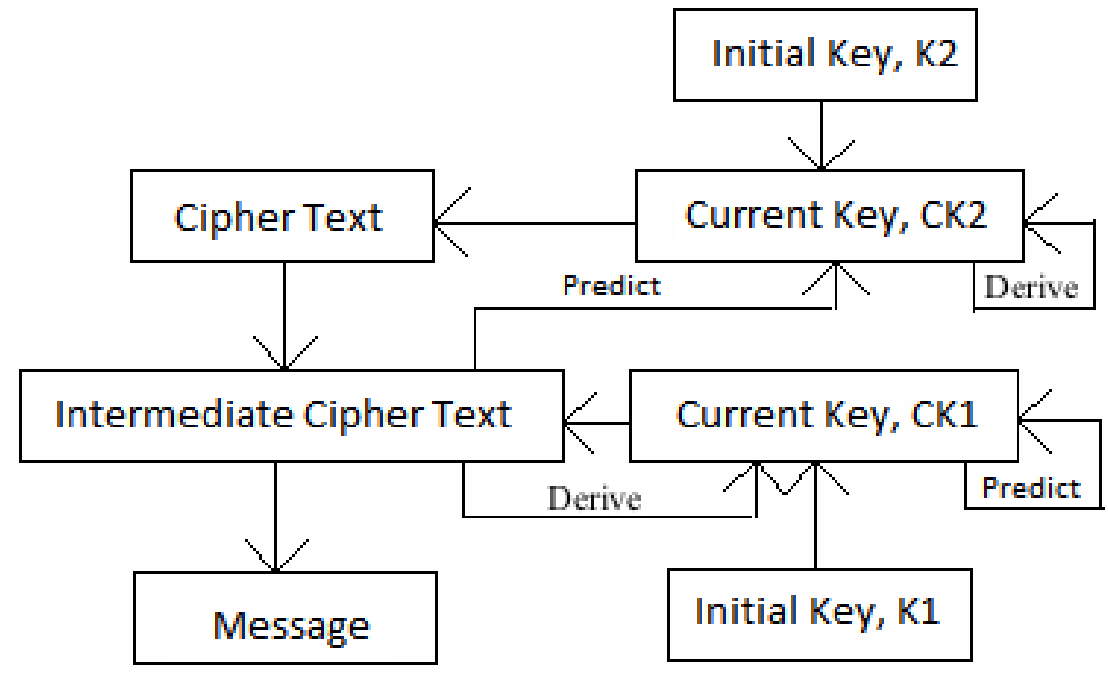


Figure 4.5: Overview of the decryption process — HIDE

4.4.4 Integrity Check

The final keys (say $K_{a(n+1)}$ and $K_{b(n+1)}$) in each round of encryption will act as the integrity check parameter. In this approach, the 128-bit key $K_{a(n+1)}$ will be concatenated with the 128-bit key $K_{b(n+1)}$ to form a 256-bit message digest, which is used for integrity check.

4.5 Example

4.5.1 Encryption and Key Generation

In this section the two rounds of the encryption process will be explained with a generic example. Each round used different initial key (K_{a0} and K_{b0}). Further, the message (M) is segmented into different blocks. The first block of the plain text message is encrypted with a key combination of K_{a0} (the present key) and K_{a1} (the currently derived key) at Round 1 that generates the first ICT block.

Let the message blocks be:

$$M = M_0, M_1, M_2, M_3, ..., M_n \tag{4.1}$$

Round − 1 : Encrypting the first block of the message: The message block will be encrypted chunk by chunk. Before the XOR operation, bit positions $0, 2, 3, 5$ will be inverted in the first chunk of an initial key. Then, the first chunk in the first block of the message will be XORed with the initial key's first chunk (which is after the inversion).

$$C'_{K_{a0},0} = \gamma(C_{K_{a0},0}) \tag{4.2}$$

$$C_{ICT_0,0} = E_{C'_{K_{a0},0}}(C_{M_0,0}) \tag{4.3}$$

Once again, bits $2, 4, 5$ and 7 in the ICT will be inverted before the key generation process and the output will be used to generate a key to encrypt the next chunk as well as the block.

$$C'_{ICT_0,0} = \gamma(C_{ICT_0,0}) \tag{4.4}$$

Each bit of the key, $C_{K_{a1},0}$, is generated from the previous key, $C'_{K_{a0},0}$, whereas, n bits in $C'_{ICT_0,0}$ chooses a bit in $C'_{K_{a0},0}$ to produce the new key, $C_{K_{a1},0}$. In general, a current key, K_{al}, is generated from the prediction of a previous ICT block, ICT_{l-1}, and the derivation of the previous key, $K_{a(l-1)}$, in Round 1. Every first chunk in the message block is directly XORed with the inversion of the first chunk in the key, whereas the 2^{nd} to 15^{th} chunk in the message will be encrypted with an inversion of the XORed value of the next chunk key with next block's chunk's key.

Key Generation: for initial chunks only

$$C_{K_{al},0} = \rho(C'_{ICT_{l-1},0}[\delta(C'_{K_{a(l-1)},0})]) \tag{4.5}$$

For the 2^{nd} to 15^{th} chunks

$$C_{K_{al},1} = [C_{K_{al},1}]XOR[C_{K_{a(l+1)l},0}] \tag{4.6}$$

...

$$C_{K_{al},15} = [C_{K_{al},15}]XOR[C_{K_{a(l+1)l},14}] \tag{4.7}$$

The generic equation for encryption in Round 1 is as follows:

$$ICT_0 = E_{K_{a0}}(M_0), ICT_1 = E_{K_{a1}}(M_1), ..., ICT_n = E_{K_{an}}(M_n) \tag{4.8}$$

Figure 4.6 is an example diagram for the encryption and integrity check processes. For this example, it is assume that the message contains only two blocks and that

each one has only two chunks (the typical block size used in this example is 16-bits, which is composed of two 8-bit chunks). The initial key, K_{a0} (10011010 11011011), is encrypted with the message (00101101 00110011 10111011 01000011) which produces, in Round 1, the ICT (00000011 11010010 10000001 01001111). Then, in Round 1, the ICT block is used to predict a derivation bit in a key block, whereas in Round 2 a key block predicts the derivation bit in the ICT block. Further, in Round 2, the cipher text (11110011 01000111 00111000 10111101) will be obtained by encrypting the ICT with the key block K_{b0} (10111011 00101100). Finally, the concatenation of the last derived keys, K_{a2} and K_{b2}, will act as the message digest, 01000111 11100010 10111001 11001111.

$Round-2$: In this Round, the encryption process is the same as in Round 1. But in the key generation process, the prediction and derivation functions are interchanged between an ICT block and a key. An initial key (K_{b0}) encrypts the first block of the ICT block. To encrypt subsequent blocks, a current key (K_{bl}) is generated from the prediction of a previous Key block ($K_{b(l-1)}$) and the derivation of a previous ICT block (ICT_{l-1}). In Round 2, the bits chosen to invert are different form Round 1. For a key bits $1, 4, 6$ and 7 are inverted before encryption, whereas $0, 1, 3$ and 6 are inverted for the ICT chunks (after the encryption, but before the key generation process).

Encrypting first block of the message:

$$C'_{K_{b0},0} = \gamma(C_{K_{b0},0}) \tag{4.9}$$

$$C_{CT_0,0} = E_{C'_{K_{b0},0}}(C_{ICT_0,0}) \tag{4.10}$$

$$C'_{ICT_0,0} = \gamma(C_{ICT_0,0}) \tag{4.11}$$

Key Generation: for initial chunks only

$$C_{K_{bl},0} = \rho(C'_{K_{b(l-1)},0}[\delta(C'_{ICT_{l-1},0})]) \tag{4.12}$$

For the 2^{nd} to 15^{th} chunks

$$C_{K_{bl},1} = [C_{K_{bl},1}]XOR[C_{K_{b(l+1)l},0}] \tag{4.13}$$

...

$$C_{K_{bl},15} = [C_{K_{bl},15}]XOR[C_{K_{b(l+1)l},14}] \tag{4.14}$$

The generic equation for encryption in Round 2 is as follows:

$$CT_0 = E_{K_{b0}}(ICT_0), CT_1 = E_{K_{b1}}(ICT_1), ..., CT_n = E_{K_{bn}}(ICT_n) \tag{4.15}$$

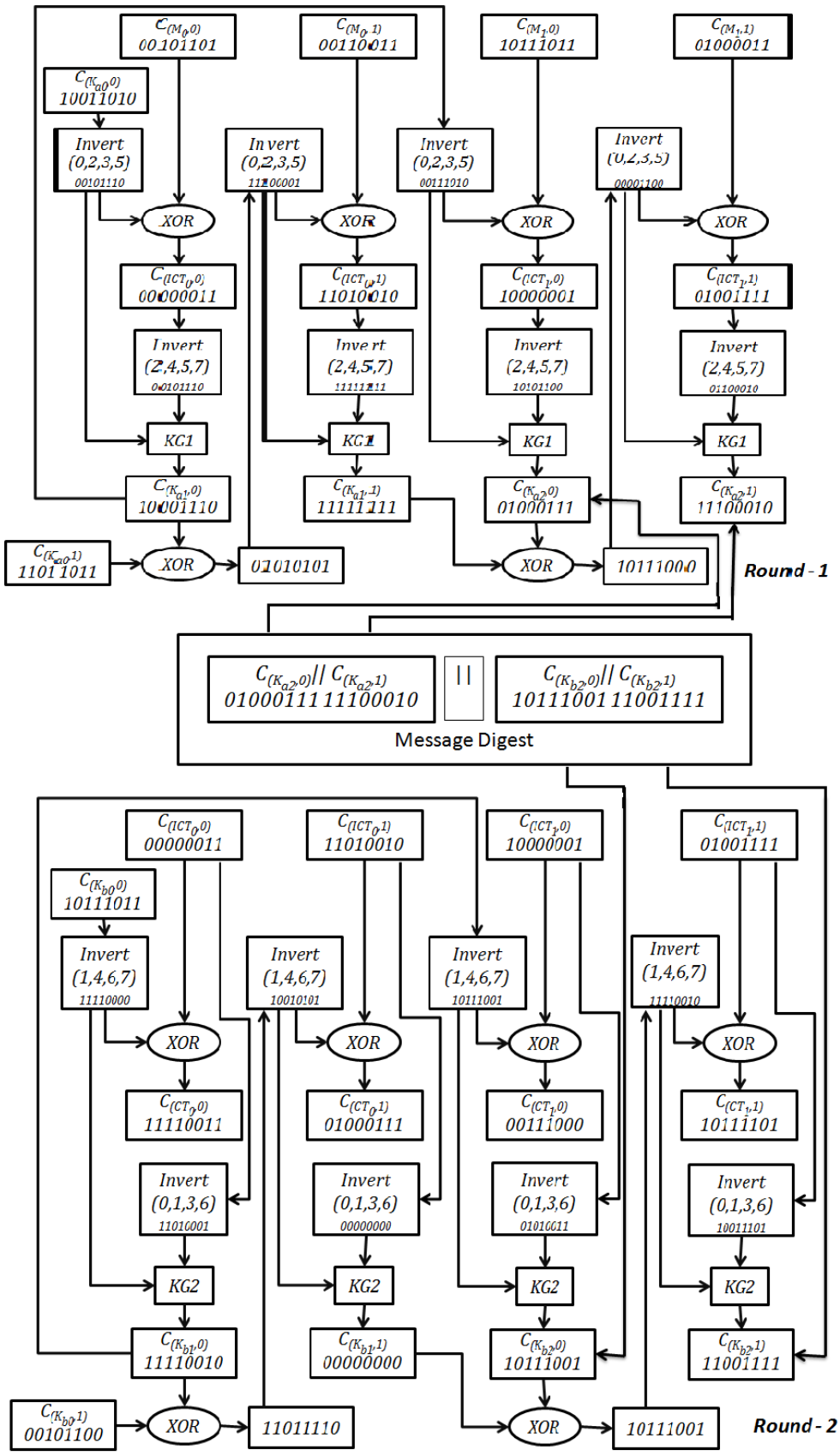


Figure 4.6: An example of the Encryption and Integrity Check Process

4.5.2 Message Digest

The final keys generated from each round will be concatenated to yield the 256-bit Message Digest.

$$MD = K_{a(n+1)}||K_{b(n+1)} \tag{4.16}$$

Chapter 5

Implementation

In this thesis, Java is used as a primary language to implement the proposed algorithms, namely, S-SCARS and HIDE. The source code for the primary functions (expansion, bit flipping, prediction and derivation) is included in this section.

5.1 S-SCARS

There are three major operations (expansion, bit flipping and compression) accomplished in S-SCARS. The bit flipping is common for both encryption and decryption. The expansion function is called while encrypting the message, whereas the compression is performed on the message during decryption.

5.1.1 Expansion

An expansion function involves doubling the message size, which inserts an integrity check as well.

```
for (int x = 0, y = 0; y < 2 * message.length; x = x + 4,
        y = y + 8) {
                expand[y + 0] = r1_xor[x + 0];
                expand[y + 1] = r1_xor[x + 0] ^ r1_xor[x + 1];
                expand[y + 2] = r1_xor[x + 3];
                expand[y + 3] = r1_xor[x + 1];
                expand[y + 4] = r1_xor[x + 2];
                expand[y + 5] = r1_xor[x + 3] ^ r1_xor[x + 2];
                expand[y + 6] = r1_xor[x + 1];
                expand[y + 7] = r1_xor[x + 3];
        }
```

5.1.2 Bit Flipping

A bit flipping function provides an additional bond to the expansion function. The key generation process uses the bit flipping function as well.

```
for(int i=0;i<2*message.length; i= i+2)
        {
                if(expand[i+0] == 0 && expand[i+1] == 0){
                        if(expand[(i+2)%128] == 0){
                                expand[(i+2)%128]=1;
                }
                        else{
                                expand[(i+2)%128]=0;
                        }
                }
                if(expand[i+0] == 0 && expand[i+1] == 1){
                        if(expand[(i+3)%128] == 0){
                                expand[(i+3)%128]=1;
                }
                        else{
                                expand[(i+3)%128]=0;
                        }
                }
                if(expand[i+0] == 1 && expand[i+1] == 0){
                        if(expand[(i+4)%128] == 0){
                                expand[(i+4)%128]=1;
                }
                        else{
                                expand[(i+4)%128]=0;
                        }
                }
                if(expand[i+0] == 1 && expand[i+1] == 1){
                        if(expand[(i+5)%128] == 0){
                                expand[(i+5)%128]=1;
```

```
                }
                        else{
                                expand[(i+5)%128]=0;
                        }
                }
        }
```

5.1.3 Compression

An expanded message is reduced to its original size while applying the compression operation, which detects any data modification in the message as well.

```
for (int x=0, y=0; y<r2_xor.length; x=x+4, y=y+8)
        {
                if((expand[y+1]==(expand[y+0]^expand[y+4])
                        && (expand[y+5]==(expand[y+3]^
                                expand[y+7]))
                        && (expand[y+2] == expand[y+7])
                        && (expand[y+6] == expand[y+0]) )){
                r1_xor[x+0] = expand[y+0];
                r1_xor[x+1] = expand[y+3];
                r1_xor[x+2] = expand[y+4];
                r1_xor[x+3] = expand[y+7];
                }
                else{
                        System.out.println("Error");
                }
        }
```

5.2 HIDE

Prediction and derivation are two major operations in HIDE that are used to generate keys dynamically. To generate successive keys, the prediction and derivation

operations are applied alternatively to the ICT and the previous key for Rounds 1 and 2.

5.2.1 Round 1

In Round 1, successive keys are derived through predicting the ICT block.

```
for (int k = 0; k < (Message.length); k = k + 128) {
        for (int z = 0; z < key.length; z = z + 1) {
                key[z] = successiveKey[z];
        }
        for (int z = 0; z < 128; z = z + 8) {
                if (key[z] == 0) {
                        key[z] = 1;
                } else {
                        key[z] = 0;
                }
                if (key[z + 2] == 0) {
                        key[z + 2] = 1;
                } else {
                        key[z + 2] = 0;
                }
                if (key[z + 3] == 0) {
                        key[z + 3] = 1;
                } else {
                        key[z + 3] = 0;
                }
                if (key[z + 5] == 0) {
                        key[z + 5] = 1;
                } else {
                        key[z + 5] = 0;
                }
        }
        for (int i = k; i < (128 + k); i = i + 8) {
```

```
toExpansion[i] = Message[i] ^
        key[(i + (0 + (k / 16))) % 128];
toExpansion[i + 1] = Message[i + 1]
        ^ key[(i + (1 + (k / 16))) % 128];
toExpansion[i + 2] = Message[i + 2]
        ^ key[(i + (2 + (k / 16))) % 128];
toExpansion[i + 3] = Message[i + 3]
        ^ key[(i + (3 + (k / 16))) % 128];
toExpansion[i + 4] = Message[i + 4]
        ^ key[(i + (4 + (k / 16))) % 128];
toExpansion[i + 5] = Message[i + 5]
        ^ key[(i + (5 + (k / 16))) % 128];
toExpansion[i + 6] = Message[i + 6]
        ^ key[(i + (6 + (k / 16))) % 128];
toExpansion[i + 7] = Message[i + 7]
        ^ key[(i + (7 + (k / 16))) % 128];

for (int z = i; z < (i + 8); z = z + 1) {
        Encrypt_IntermediateCipherText[z]
                = toExpansion[z];
}
if (toExpansion[i + 2] == 0) {
        toExpansion[i + 2] = 1;
} else {
        toExpansion[i + 2] = 0;
}
if (toExpansion[i + 4] == 0) {
        toExpansion[i + 4] = 1;
} else {
        toExpansion[i + 4] = 0;
}
if (toExpansion[i + 5] == 0) {
```

```
            toExpansion[i + 5] = 1;
        } else {
            toExpansion[i + 5] = 0;
        }
        if (toExpansion[i + 7] == 0) {
            toExpansion[i + 7] = 1;
        } else {
            toExpansion[i + 7] = 0;
        }
        for (int a = 0; a < 8; a = a + 1) {
        if (toExpansion[i + (a + 0) % 8] == 0
        && toExpansion[i + (a + 1) % 8] == 0
        && toExpansion[i + (a + 2) % 8] == 0) {
        successiveKey[i % 128]
        = key[(i + 7) % 128] ^ key[(i + 1) % 128];
        } else if (toExpansion[i + (a + 0) % 8] == 0
        && toExpansion[i + (a + 1) % 8] == 0
        && toExpansion[i + (a + 2) % 8] == 1) {
        successiveKey[(i + 1) % 128]
        = key[(i + 0) % 128] ^ key[(i + 2) % 128];
        } else if (toExpansion[i + (a + 0) % 8] == 0
        && toExpansion[i + (a + 1) % 8] == 1
        && toExpansion[i + (a + 2) % 8] == 0) {
        successiveKey[(i + 2) % 128]
        = key[(i + 1) % 128] ^ key[(i + 3) % 128];
        } else if (toExpansion[i + (a + 0) % 8] == 0
        && toExpansion[i + (a + 1) % 8] == 1
        && toExpansion[i + 2] == (a + 1) % 8) {
        successiveKey[(i + 3) % 128]
        = key[(i + 2) % 128] ^ key[(i + 4) % 128];
        } else if (toExpansion[i + (a + 0) % 8] == 1
        && toExpansion[i + (a + 1) % 8] == 0
```

```
                    && toExpansion[i + (a + 2) % 8] == 0) {
                    successiveKey[(i + 4) % 128]
                    = key[(i + 3) % 128] ^ key[(i + 5) % 128];
                    } else if (toExpansion[i + (a + 0) % 8] == 1
                    && toExpansion[i + (a + 1) % 8] == 0
                    && toExpansion[i + (a + 2) % 8] == 1) {
                    successiveKey[(i + 5) % 128]
                    = key[(i + 4) % 128] ^ key[(i + 6) % 128];
                    } else if (toExpansion[i + (a + 0) % 8] == 1
                    && toExpansion[i + (a + 1) % 8] == 1
                    && toExpansion[i + (a + 2) % 8] == 0) {
                    successiveKey[(i + 6) % 128]
                    = key[(i + 5) % 128] ^ key[(i + 7) % 128];
                    } else if (toExpansion[i + (a + 0) % 8] == 1
                    && toExpansion[i + (a + 1) % 8] == 1
                    && toExpansion[i + (a + 2) % 8] == 1) {
                    successiveKey[(i + 7) % 128]
                    = key[(i + 6) % 128] ^ key[(i + 0) % 128];
                    }
            }
key[(i+8)%128] = key[(i+8)%128]^successiveKey[(i+0)%128];
key[(i+9)%128] = key[(i+9)%128]^successiveKey[(i+1)%128];
key[(i+10)%128] = key[(i+10)%128]^successiveKey[(i+2)%128];
key[(i+11)%128] = key[(i+11)%128]^successiveKey[(i+3)%128];
key[(i+12)%128] = key[(i+12)%128]^successiveKey[(i+4)%128];
key[(i+13)%128] = key[(i+13)%128]^successiveKey[(i+5)%128];
key[(i+14)%128] = key[(i+14)%128]^successiveKey[(i+6)%128];
key[(i+15)%128] = key[(i+15)%128]^successiveKey[(i+7)%128];
}
}
```

5.2.2 Round 2

In Round 2, successive keys are derived through predicting the previous key block.

```
for (int k = 0; k < (Message.length); k = k + 128) {
        for (int z = 0; z < key2.length; z = z + 1) {
                key2[z] = successiveKey2[z];
        }
        for (int z = 0; z < 128; z = z + 8) {
                if (key2[z + 1] == 0) {
                        key2[z + 1] = 1;
                } else {
                        key2[z + 1] = 0;
                }
                if (key2[z + 4] == 0) {
                        key2[z + 4] = 1;
                } else {
                        key2[z + 4] = 0;
                }
                if (key2[z + 6] == 0) {
                        key2[z + 6] = 1;
                } else {
                        key2[z + 6] = 0;
                }
                if (key2[z + 7] == 0) {
                        key2[z + 7] = 1;
                } else {
                        key2[z + 7] = 0;
                }
        }
        for (int i = k; i < (128 + k); i = i + 8) {
        toExpansion2[i] = toExpansion[i]
                        ^ key2[(i + (0 + (k / 16))) % 128];
        toExpansion2[i + 1] = toExpansion[i + 1]
```

```
                ^ key2[(i + (1 + (k / 16))) % 128];
toExpansion2[i + 2] = toExpansion[i + 2]
                ^ key2[(i + (2 + (k / 16))) % 128];
toExpansion2[i + 3] = toExpansion[i + 3]
                ^ key2[(i + (3 + (k / 16))) % 128];
toExpansion2[i + 4] = toExpansion[i + 4]
                ^ key2[(i + (4 + (k / 16))) % 128];
toExpansion2[i + 5] = toExpansion[i + 5]
                ^ key2[(i + (5 + (k / 16))) % 128];
toExpansion2[i + 6] = toExpansion[i + 6]
                ^ key2[(i + (6 + (k / 16))) % 128];
toExpansion2[i + 7] = toExpansion[i + 7]
                ^ key2[(i + (7 + (k / 16))) % 128];

        for (int z = i; z < (i + 8); z = z + 1) {
                CipherText[z] = toExpansion2[z];
        }
        if (toExpansion[i + 0] == 0) {
                toExpansion[i + 0] = 1;
        } else {
                toExpansion[i + 0] = 0;
        }
        if (toExpansion[i + 1] == 0) {
                toExpansion[i + 1] = 1;
        } else {
                toExpansion[i + 1] = 0;
        }
        if (toExpansion[i + 3] == 0) {
                toExpansion[i + 3] = 1;
        } else {
                toExpansion[i + 3] = 0;
        }
```

```
if (toExpansion[i + 6] == 0) {
        toExpansion[i + 6] = 1;
} else {
        toExpansion[i + 6] = 0;
}
for (int a = 0; a < 8; a = a + 1) {
if (key2[(i + (a + 0) % 8) % 128] == 0
&& key2[(i + (a + 1) % 8) % 128] == 0
&& key2[(i + (a + 2) % 8) % 128] == 0) {
successiveKey2[i % 128]
= toExpansion[i + 7] ^ toExpansion[i + 1];
} else if (key2[(i + (a + 0) % 8) % 128] == 0
&& key2[(i + (a + 1) % 8) % 128] == 0
&& key2[(i + (a + 2) % 8) % 128] == 1) {
successiveKey2[(i + 1) % 128]
= toExpansion[i + 0] ^ toExpansion[i + 2];
} else if (key2[(i + (a + 0) % 8) % 128] == 0
&& key2[(i + (a + 1) % 8) % 128] == 1
&& key2[(i + (a + 2) % 8) % 128] == 0) {
successiveKey2[(i + 2) % 128]
= toExpansion[i + 1] ^ toExpansion[i + 3];
} else if (key2[(i + (a + 0) % 8) % 128] == 0
&& key2[(i + (a + 1) % 8) % 128] == 1
&& key2[(i + (a + 2) % 8) % 128] == 1) {
successiveKey2[(i + 3) % 128]
= toExpansion[i + 2] ^ toExpansion[i + 4];
} else if (key2[(i + (a + 0) % 8) % 128] == 1
&& key2[(i + (a + 1) % 8) % 128] == 0
&& key2[(i + (a + 2) % 8) % 128] == 0) {
successiveKey2[(i + 4) % 128]
= toExpansion[i + 3] ^ toExpansion[i + 5];
} else if (key2[(i + (a + 0) % 8) % 128] == 1
```

```
                    && key2[(i + (a + 1) % 8) % 128] == 0
                    && key2[(i + (a + 2) % 8) % 128] == 1) {
                    successiveKey2[(i + 5) % 128]
                    = toExpansion[i + 4] ^ toExpansion[i + 6];
                    } else if (key2[(i + (a + 0) % 8) % 128] == 1
                    && key2[(i + (a + 1) % 8) % 128] == 1
                    && key2[(i + (a + 2) % 8) % 128] == 0) {
                    successiveKey2[(i + 6) % 128]
                    = toExpansion[i + 5] ^ toExpansion[i + 7];
                    } else if (key2[(i + (a + 0) % 8) % 128] == 1
                    && key2[(i + (a + 1) % 8) % 128] == 1
                    && key2[(i + (a + 2) % 8) % 128] == 1) {
                    successiveKey2[(i + 7) % 128]
                    = toExpansion[i + 6] ^ toExpansion[i + 0];
                    }
            }
key2[(i+8)%128] = key2[(i+8)%128]^successiveKey2[(i+0)%128];
key2[(i+9)%128] = key2[(i+9)%128]^successiveKey2[(i+1)%128];
key2[(i+10)%128] = key2[(i+10)%128]^successiveKey2[(i+2)%128];
key2[(i+11)%128] = key2[(i+11)%128]^successiveKey2[(i+3)%128];
key2[(i+12)%128] = key2[(i+12)%128]^successiveKey2[(i+4)%128];
key2[(i+13)%128] = key2[(i+13)%128]^successiveKey2[(i+5)%128];
key2[(i+14)%128] = key2[(i+14)%128]^successiveKey2[(i+6)%128];
key2[(i+15)%128] = key2[(i+15)%128]^successiveKey2[(i+7)%128];
}
}
```

5.3 Hardware Implementation

An encryption process of the proposed algorithms are implemented using Xilinx Spartan-6 FPGA. A summary of resource utilization is presented in the Table 5.1. Figure 5.1 and 5.2 are hardware implementation of SCARS and HIDE respectively.

Table 5.1: Summary of Hardware Implementation

	SCARS	HIDE [32-bit I/O]
Number of Slices	124	69
Number of LUTs	269	173
Number of bounded IOBs	255	144
Average Fanout of Non-Clock Nets	3.12	3.78

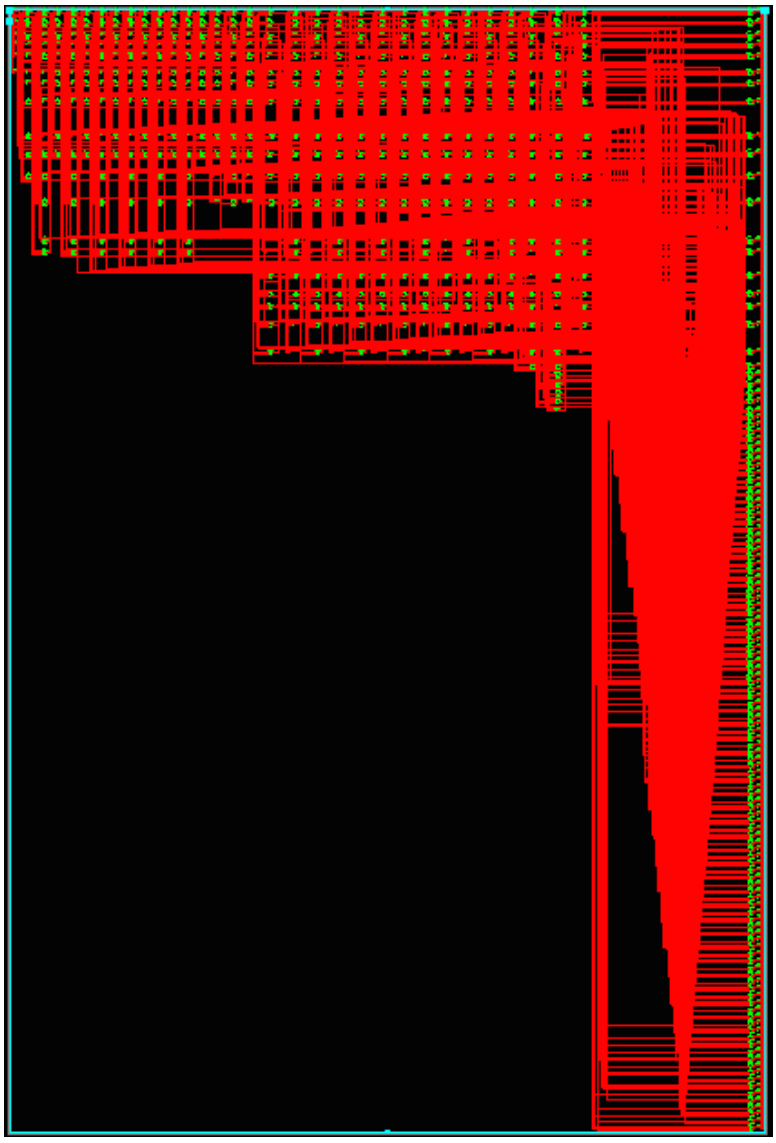


Figure 5.1: Implementation of SCARS

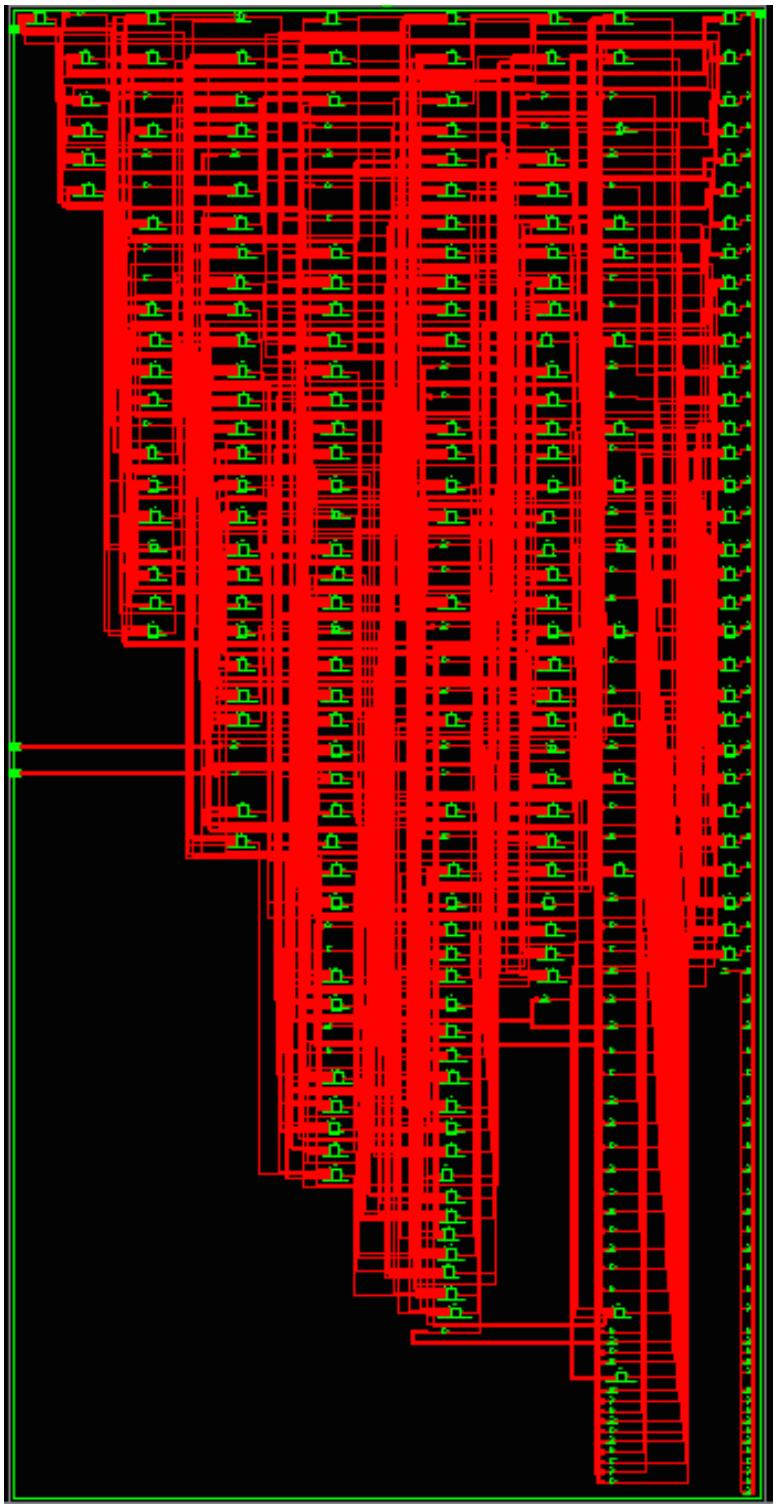


Figure 5.2: Implementation of HIDE

Chapter 6

Evaluation

6.1 S-SCARS

A resource-constrained RFID system has a limited number of gates for implementing advanced cryptographic algorithms, which leaves the system vulnerable to different attacks. In this section, the qualitative security analysis along with the cryptanalysis and performance evaluation will be discussed in detail.

6.1.1 Security Analysis

In a *de-synchronization attack*, an attacker interrupts the communication between two parties and either modifies or drops the message [2]. As different keys are used that are synchronized between the tag and server without the key exchange, dropping or modifying the message will lead to de-synchronization of the key between them. To avoid such a situation, an acknowledgment code n' is used (which is derived from the pre-shared key) from the server to the tag. As the acknowledgment code is in plain text, the tag simply compares it without any complex operation and accepts it only if both the received and existing (tag generated) acknowledgment codes are same. If the message is dropped, the key will be desynchronized automatically, which leads to the server notifying the administrator to reset the key. Though it leads to overhead for the administrator (to reset the key), this approach ensures high security. Since integrity is provided as part of the encryption mechanism, the server can identify a modified message easily and notify the administrator automatically to reset the key.

In a *replay attack*, an attacker eavesdrop the current information that can be used later to pretend to be an authorized entity [39]. To avoid an attack using an eavesdropped message for later communication, the key is changed for every transmission, which results in avoiding the replay attack. A *denial of service attack* involves the

attacker blocking the service either temporarily or permanently between the communication parties [39]. By establishing communication between the server and the tag with an authentication code and terminating by updating the nonce (also used for an authentication) at both end makes the server aware of the denial of service attack.

A *man in the middle attack* allows an attacker to eavesdrop, modify or deny the service between the communicating parties [19]. As discussed earlier, none of this can be launched on the proposed approach because of the use of key changes for each communication. The server can identify an attacker's intervention easily while communicating with the tag. A *data manipulation attack* is identified at the server end while decrypting the received message [19].

Forward secrecy assures that previous keys are secure even after revealing the current key, while *backward secrecy* assures that the next key cannot be generated from the current key [24]. With the proposed approach, an attacker can neither retrieve information nor generate a next key from the current one, which prevents the attacker from being able to generate the next or previous keys.

6.1.2 Cryptanalysis

This section presents the qualitative cryptanalysis (chosen-plain text attack, chosen-cipher text attack and differential attack) for using this approach on RFID systems [37]. Each attack is given a short description followed by a detailed analysis. On one hand, in the chosen-plain text attack, an attacker assumes that a message is in plain text that is encrypted to obtain a corresponding cipher text, which will be compared with captured cipher text. On the another hand, an attacker recovering the plain text from the chosen cipher text to obtain the secret key. The proposed approach changes the key for every communication, resulting in different cipher texts even though the given message is the same request from the server. By updating the nonce to offer authentication, even though the key is the same for the current communication, the message (including the nonce with its random bits) is different. Thus, it assures that the proposed approach is resistant to both the chosen-plain text and chosen-cipher text attacks.

A differential attack compares the difference in an input value with the output to obtain the actual message, which is a form of chosen-plain text attack [37]. As

discussed earlier, a change in key for each communication or changing the message for a current communication secures this approach from differential attack. Furthermore, even though the second key is derived from the first one, the expansion and bit flipping functions modify the bit positions to ensure no relation between the two rounds of XOR operation.

The proposed approach was tested under different combinations of input messages. The sample sets chosen to test the bit flipping were 16-bit and 128-bit inputs. A result in the output cipher text was randomly flipped in the bit flipping function though the first bit in the message wasn't changed in the expansion function. In addition to that, though the message is expanded byte-by-byte, bit flipping assures connection between the bytes and offers addition support to integrity. In the future work, extensive cryptanalysis will be performed to evaluate the expansion function to make necessary changes on changing bit positions.

6.1.3 Security Analysis using Cryptool

In addition to the above-mentioned attacks, the proposal was tested using a cryptanalysis tool called Cryptool [1]. The security analysis of the proposed approach was compared with existing algorithms such as, AES, DES, Triple DES, RC4, etc. (implementation available as a built-in option in Cryptool). Figure 6.1, 6.2 and 6.3 are presented with three different scenarios, which are based on the cryptanalysis tests (entropy test, periodicity test, frequency test, poker test, run test and the serial test) [41][5][36].

The results from the analysis show that proposed algorithm passes the above mentioned tests that were carried out and that the security level is no lower than the example algorithms, which were used.

6.1.4 Performance Evaluation

In this section, a summary of the estimated Gate Estimate (GE) used is presented in Table 6.1 and is compared as well with the other algorithms [32]. It can be seen that the proposed algorithm is more efficient since it requires fewer logic gates, and hence, fewer operations to be performed.

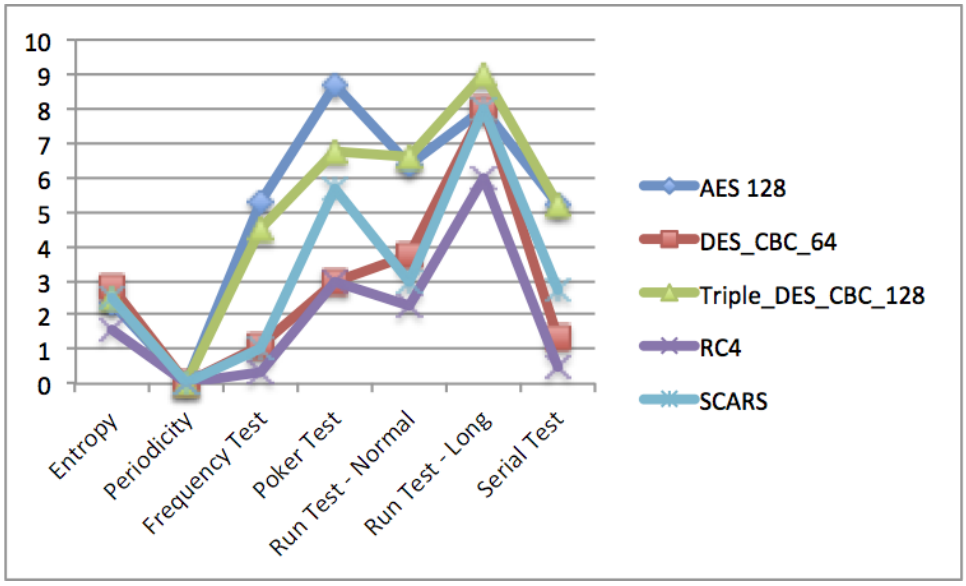


Figure 6.1: Sample 1 — S-SCARS

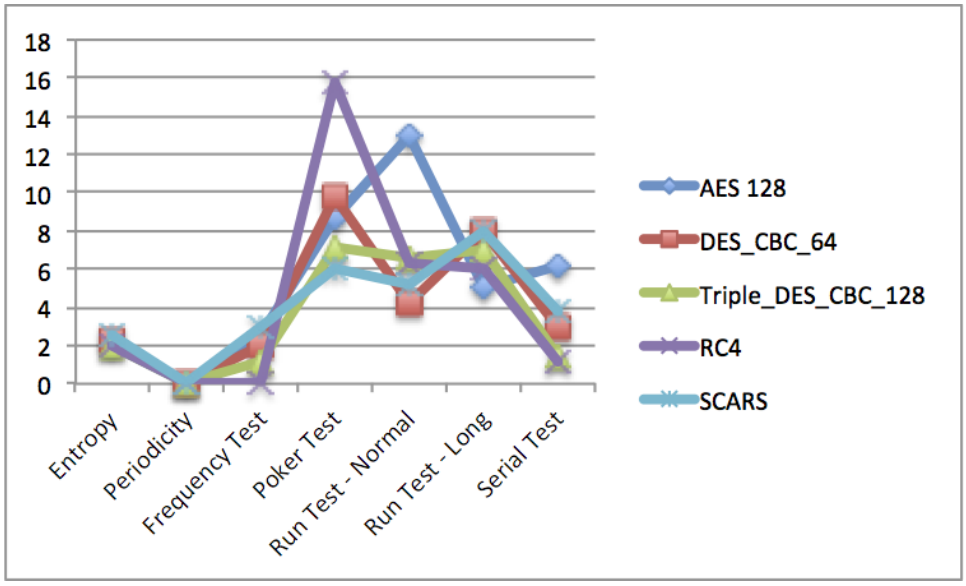


Figure 6.2: Sample 2 — S-SCARS

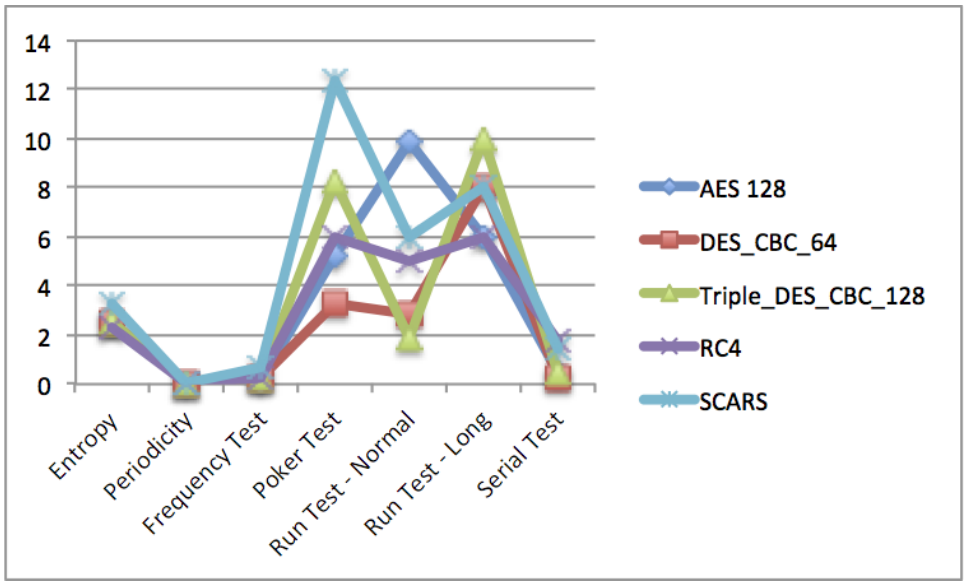


Figure 6.3: Sample 3 — S-SCARS

Table 6.1: GE Comparison Table

	SCARS [32]	RBS [20]	AES [14]	PRESENT [34]	Grain [15]
# of gates	527 GE	1920 GE	3200 GE	2332 GE	1857 GE
Block size	64-128 bits	132 bits	128 bits	64 bits	1 bit

The proposal uses four 32-bit XOR gates, one NOT gate and one 2-bit comparator that is implemented using two 1-bit comparator circuits. Each 1-bit comparator is implemented using six 2 input NAND gates, and combined using two additional NAND gates. Each 2-bit XOR is considered to be implemented using four NAND gates. Therefore, there are approximately 527 two-input NAND gates in the implementation. However, this is subject to further comprehensive complexity analysis.

6.2 HIDE

6.2.1 Security Analysis

Most previous approaches derive successive keys from an initial or intermediate key. This proposal has successive key derivation function, which uses both a key as well as a message to choose the next key dynamically. Every message block is encrypted with

a different key chosen from the 2^{128} possible combination keys for a given 128-bits in the key space.

As successive keys are generated based on a message and a key in any one process of either prediction or derivation, the generated key is completely dynamic. Thus, it guarantees that there would be no regular cycle in a key bit stream. The algorithm is a very simple but efficient one, which uses XOR operations to generate keys as well as encrypt a message. It can withstand a variety of attacks (chosen-plain text attack, chosen-cipher attack, differential attack, linear attack and distinguishing attack) [16] [44] [33].

6.2.2 Cryptanalysis

A short description of possible attacks and their analyses are explained in this section. Chosen-plain text and chosen-cipher text attacks are targeted on symmetric key encryption schemes. Both attacks work on the principle of choosing a piece of information for retrieving an original message for a given cipher text or a cipher text for a given plain text. In this approach, since the key is changed for every block encrypted, chosen-plain text and chosen-cipher text attacks are hard to launch. A differential attack compares an input value with an output value to obtain a possible key. Since this proposal relies on both a key and an ICT, and the key is chosen from a strong key, a differential attack is difficult to implement. In linear cryptanalysis, an equation is formed from plain text and cipher text, which is equated to possible key bits to reveal some information about the message. To avoid this issue, the key generation process is switched between prediction and derivation techniques. A distinguishing attack focuses on stream ciphers, which compare a given sequence of values to check the randomness. A dynamically generated key ensures that there will be no relation between the current and previous key, so launching a distinguishing attack is difficult.

6.2.3 Security Analysis using Cryptool

In addition to the attacks mentioned above, tests were done using a cryptanalysis tool called Cryptool [1]. As mentioned earlier, though the key generation part is in the nature of a stream cipher, the actual encryption process adopts a block cipher

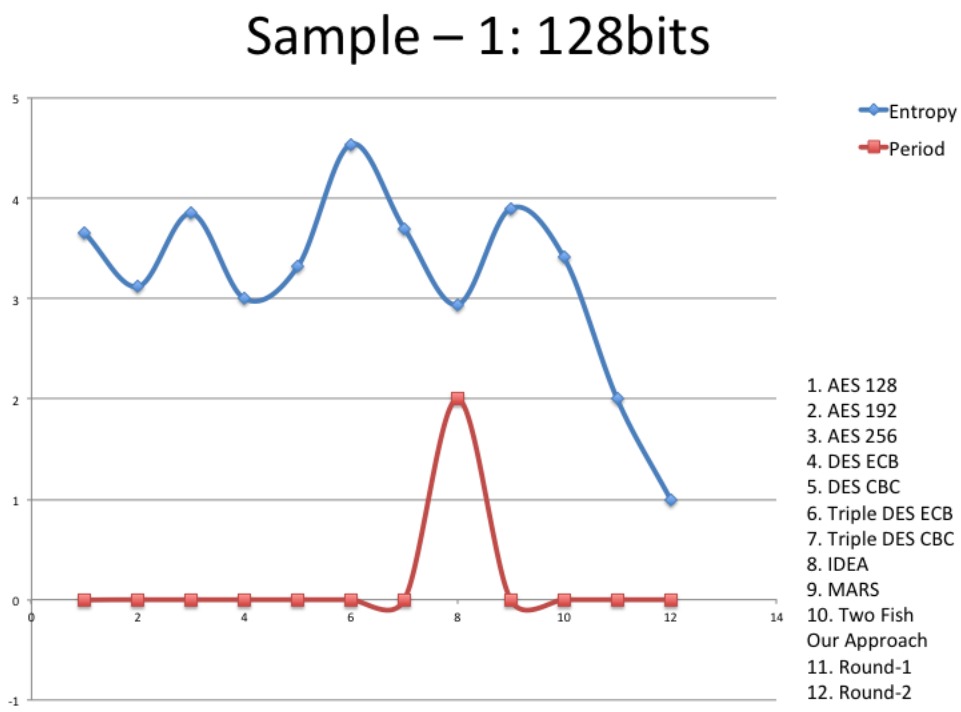


Figure 6.4: Sample 1 — HIDE

approach. The security analysis of the proposed approach was compared with existing block cipher algorithms such as AES, DES, IDEA, MARS, Twofish etc. (the implementation of which is available as a built in option in Cryptool). Graphs are presented below with five different scenarios, which are based on the cryptanalysis tests (entropy test, periodicity test, frequency test, poker test, run test and serial test) [41] [5] [36]. The results from this analysis show that the proposed algorithm passes the above mentioned tests and that the security level is no lower than the pool of example algorithms used. Figures 6.4, 6.5, 6.6, 6.7, 6.8 display the sample graphs, which were taken from the security analysis with different sample inputs.

The proposed algorithm provides non-linearity in the key cycle, which is achieved by using an intermediate cipher text block. However, using cipher text to encrypt a message is vulnerable, so two rounds of encryption were designed with an ICT as the second round key generator instead of the cipher text. Applying key generation from a key for the first round and an ICT for the second round ensures a dynamic property in successive keys.

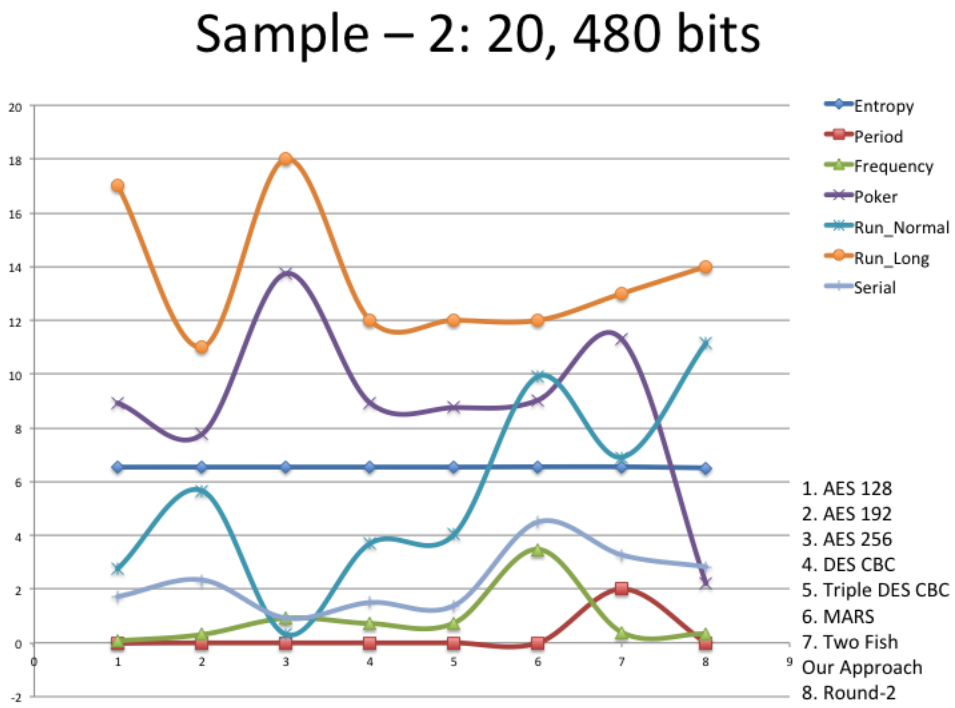


Figure 6.5: Sample 2 — HIDE

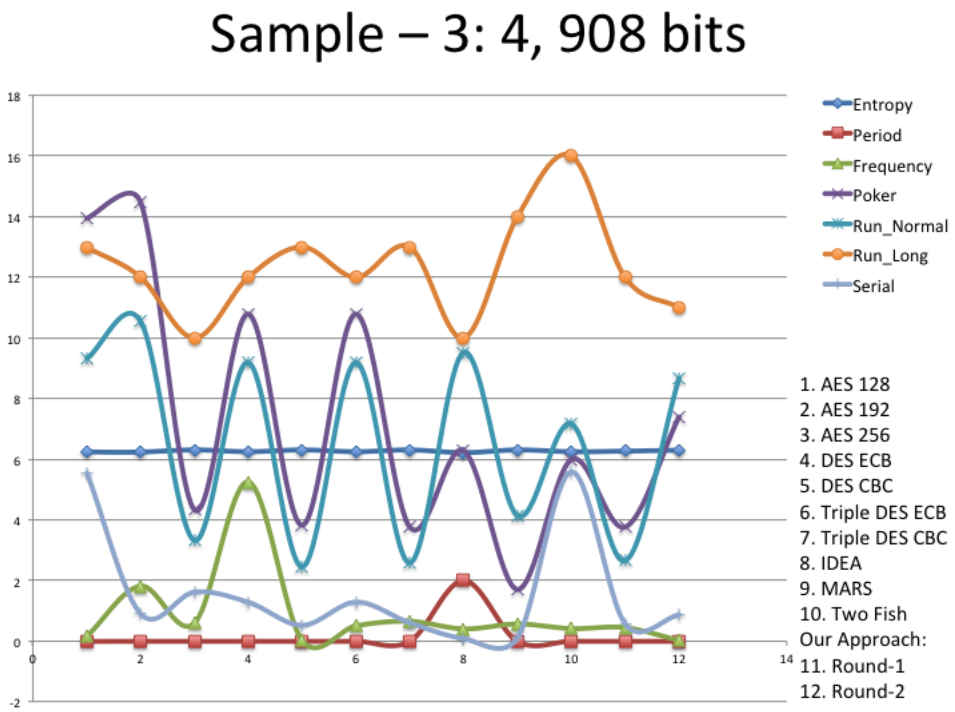


Figure 6.6: Sample 3 — HIDE

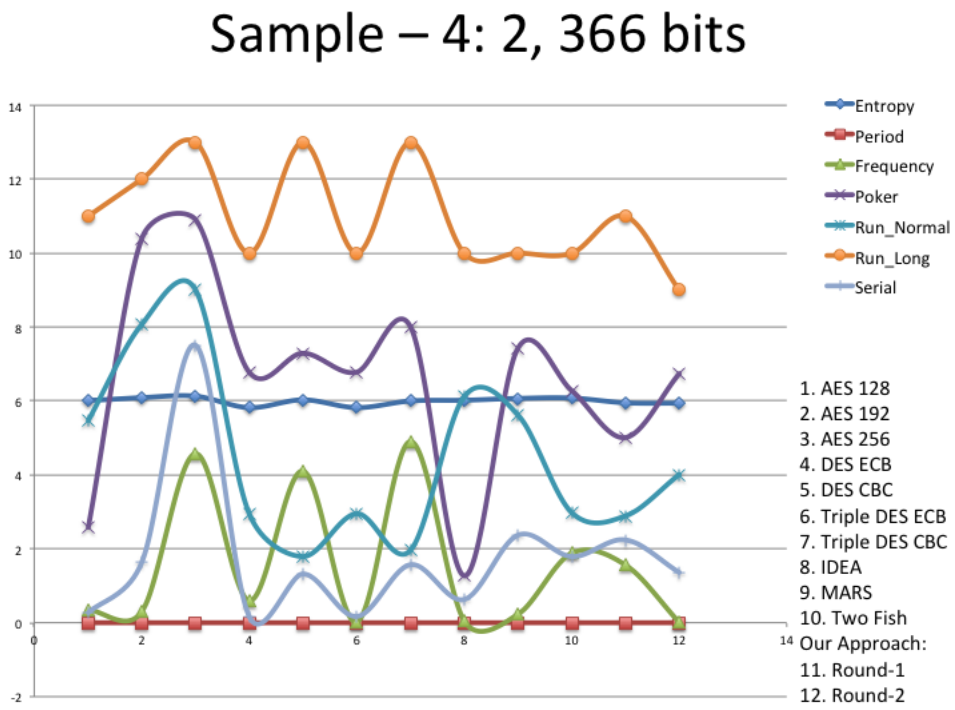


Figure 6.7: Sample 4 — HIDE

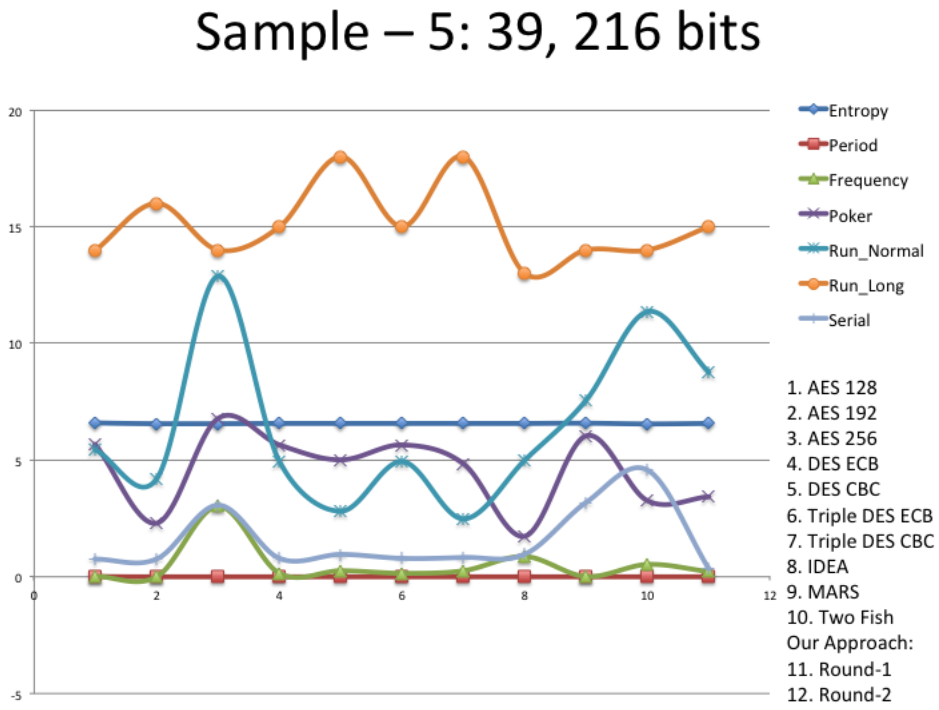


Figure 6.8: Sample 5 — HIDE

6.2.4 Performance Evaluation

The performance analysis of this approach has yet to be tested, but it is expected to be computationally efficient, which is based on the following claims. Initially, the computation is required only for the encryption part, but the integrity check does not require any additional computation process. As the key generation process is a bitwise operation, it is the only difficulty in this approach. The number of operations required per bit is only fourteen, which includes encryption (or decryption), an integrity check and key generation operations. A qualitative analysis proves that an increase in performance would require an increase in the hardware component, which may not be appropriate depending on the requirements of the application. However, future work on the key generation part is expecting to reduce the hardware requirement, increase the computational efficiency of the proposed approach.

6.3 Discussion

The key functionalities necessary to implement confidentiality, integrity and authentication in a single symmetric key encryption without using any external algorithms were described. The combining of the different security goals into one algorithm will reduce the overall computation cost required by the cryptosystem, which will improve the efficiency of the proposed approaches. Hashing is not used for integrity nor are there external algorithms for authentication. Instead, simple random bit generation is introduced for authentication (S-SCARS only). Consequently, the computational cost for the proposed protocol will be less than the computational cost of performing encryption with hashing and authentication, which can be summarized as:

$Cost(Authentication{+}Encryption{+}Integrity)^1 < Cost(Encryption{+}Integrity)^2 < Cost(Signature + Encryption) << [Cost(Signature) + Cost(Encryption)]$ [43]

On the one hand, S-SCARS uses expansion and bit flipping functions, which ensures integrity and random number generation for authentication. On the other hand, HIDE uses the prediction and derivation combination to ensure integrity. Further, both schemes adopt XOR operations, which ensures confidentiality. In summary, S-SCARS ensures confidentiality, integrity and authentication, whereas HIDE offers

[1] S-SCARS
[2] HIDE

only confidentiality and integrity. A qualitative security analysis, including tests against standard data analysis proves that the proposed schemes are secure.

Chapter 7

Conclusion and Future Work

This thesis proposes two novel symmetric key encryption algorithms, namely, S-SCARS and HIDE. S-SCARS offers confidentiality, integrity and authentication for resource-constrained systems such as RFID without using multiple algorithms, whereas HIDE is a hybrid approach of stream and block ciphers that is designed to offer confidentiality, integrity and dynamic key generation.

In conclusion, the uniqueness of the algorithms is to achieve more than one security goal without using additional algorithms and this is accomplished with limited resources. A qualitative security analysis has been performed to evaluate the security features of the proposed approaches and is compared with standard security algorithms. Overall, the proposed algorithms use less resources with better security compared with existing protocols.

The resource utilization of HIDE was expected to be higher than S-SCARS. Surprisingly, both algorithms use comparatively the same amount of resources. In addition, the security analyses of the algorithms were tested using Cryptool, which is considered as an educational tool. Finally, proposed approaches are not available in real-time environment. For these reasons, improved security analysis, study of hardware implementation for HIDE, and real-time algorithm deployment will be conducted in the future.

Bibliography

[1] The cryptool portal @ONLINE, 2014.

[2] Zahra Ahmadian, Mahmoud Salmasizadeh, and Mohammad Reza Aref. Desynchronization attack on rapp ultralightweight authentication protocol. *Information processing letters*, 113(7):205–209, 2013.

[3] NG Bardis, AP Markovskyy, and DV Andrikou. Method for designing pseudorandom binary sequences generators on nonlinear feedback shift register(nfsr). *WSEAS Transactions on Communications*, 3(2):758–763, 2004.

[4] Lejla Batina, Jorge Guajardo, Tim Kerins, Nele Mentens, Pim Tuyls, and Ingrid Verbauwhede. Public-key cryptography for rfid-tags. In *Pervasive Computing and Communications Workshops, 2007. PerCom Workshops' 07. Fifth Annual IEEE International Conference on*, pages 217–222. IEEE, 2007.

[5] Dan Biebighauser. Testing random number generators @ONLINE, 2000.

[6] Chih-Chun Chang, Sead Muftic, and David J Nagel. Measurement of energy costs of security in wireless sensor nodes. In *Computer Communications and Networks, 2007. ICCCN 2007. Proceedings of 16th International Conference on*, pages 95–102. IEEE, 2007.

[7] Chih-Chun Chang, David J Nagel, and Sead Muftic. Balancing security and energy consumption in wireless sensor networks. In *Mobile Ad-Hoc and Sensor Networks*, pages 469–480. Springer, 2007.

[8] Xiangqian Chen, Kia Makki, Kang Yen, and N. Pissinou. Sensor network security: a survey. *Communications Surveys Tutorials, IEEE*, 11(2):52–73, Second 2009.

[9] Hung-Yu Chien. Sasi: A new ultralightweight rfid authentication protocol providing strong authentication and strong integrity. *Dependable and Secure Computing, IEEE Transactions on*, 4(4):337–340, 2007.

[10] Christophe De Cannière. Trivium: A stream cipher construction inspired by block cipher design principles. In *Information Security*, pages 171–186. Springer, 2006.

[11] Yevgeniy Dodis and Jee Hea An. Concealment and its applications to authenticated encryption. In *Advances in Cryptology EUROCRYPT 2003*, pages 312–329. Springer, 2003.

[12] Elena Dubrova, Maxim Teslenko, and Hannu Tenhunen. On analysis and synthesis of (n, k)-non-linear feedback shift registers. In *Design, Automation and Test in Europe, 2008. DATE'08*, pages 1286–1291. IEEE, 2008.

[13] Daniel Engels, Xinxin Fan, Guang Gong, Honggang Hu, and Eric M Smith. Hummingbird: ultra-lightweight cryptography for resource-constrained devices. In *Financial Cryptography and Data Security*, pages 3–18. Springer, 2010.

[14] Martin Feldhofer, Johannes Wolkerstorfer, and Vincent Rijmen. Aes implementation on a grain of sand. *IEE Proceedings-Information Security*, 152(1):13–20, 2005.

[15] T Good and M Benaissa. Hardware results for selected stream cipher candidates. *State of the Art of Stream Ciphers*, pages 191–204, 2007.

[16] Martin Hell, Thomas Johansson, and Lennart Brynielsson. An overview of distinguishing attacks on stream ciphers. *Cryptography and Communications*, 1(1):71–94, 2009.

[17] Honggang Hu and Guang Gong. Periods on two kinds of nonlinear feedback shift registers with time varying feedback functions. *International Journal of Foundations of Computer Science*, 22(06):1317–1329, 2011.

[18] Ming Hu and Yan Wang. The collision rate tests of two known message digest algorithms. In *Computational Intelligence and Security, 2009. CIS'09. International Conference on*, volume 2, pages 319–323. IEEE, 2009.

[19] Pradip M Jawandhiya, Mangesh M Ghonge, MS Ali, and JS Deshpande. A survey of mobile ad hoc network attacks. *International Journal of Engineering Science and Technology*, 2(9):4063–4071, 2010.

[20] Zahra Jeddi, Esmaeil Amini, and Magdy Bayoumi. Rbs: Redundant bit security algorithm for rfid systems. In *Computer Communications and Networks (ICCCN), 2012 21st International Conference on*, pages 1–5. IEEE, 2012.

[21] Zahra Jeddi, Esmaeil Amini, and Magdy Bayoumi. A novel authenticated encryption algorithm for rfid systems. In *Digital System Design (DSD), 2013 Euromicro Conference on*, pages 658–661. IEEE, 2013.

[22] Kutuboddin Jinabade and Krupa Rasane. Efficient implementation of hummingbird cryptographic algorithm on a reconfigurable platform. *International Journal of Engineering*, 2(7), 2013.

[23] A. Juels. Rfid security and privacy: a research survey. *Selected Areas in Communications, IEEE Journal on*, 24(2):381–394, Feb 2006.

[24] Yongdae Kim, Adrian Perrig, and Gene Tsudik. Communication-efficient group key agreement. *Trusted Information: The New Decade Challenge*, pages 229–244, 2002.

[25] Lars R Knudsen. Practically secure feistel ciphers. In *Fast Software Encryption*, pages 211–221. Springer, 1994.

[26] Yogesh Kumar, Rajiv Munjal, and Harsh Sharma. Comparison of symmetric and asymmetric cryptography with existing vulnerabilities and countermeasures. *International Journal of Computer Science and Management Studies*, 11(03), 2011.

[27] Xuhong Li, Wei Zhang, Xia Wang, and Muhai Li. Novel convertible authenticated encryption schemes without using hash functions. In *Computer Science and Automation Engineering (CSAE), 2012 IEEE International Conference on*, volume 1, pages 504–508. IEEE, 2012.

[28] Hong Lv, Jian-Xia Xie, Jun-Chu Fang, and Peng Qi. Generating of a nonlinear pseudorandom sequence using linear feedback shift register. In *ICT Convergence (ICTC), 2012 International Conference on*, pages 432–435. IEEE, 2012.

[29] Kalikinkar Mandal and Guang Gong. Probabilistic generation of good span n sequences from nonlinear feedback shift registers. *University of Waterloo*, 2012.

[30] David Molnar and David Wagner. Privacy and security in library rfid: issues, practices, and architectures. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 210–219. ACM, 2004.

[31] J. Narayanaswamy, R. V. Sampangi, and S. Sampalli. HIDE: Hybrid symmetric key algorithm for integrity check, dynamic key generation and encryption. In *Proceedings of the 1st International Conference on Information Systems Security and Privacy (ICISSP-2015)*, pages 124–131. SCITEPRESS (Science and Technology Publications, Lda.), 2015.

[32] Jayagopal Narayanaswamy, Raghav V Sampangi, and Srinivas Sampalli. Scars: Simplified cryptographic algorithm for rfid systems. In *RFID Technology and Applications Conference (RFID-TA), 2014 IEEE*, pages 32–37. IEEE, 2014.

[33] Chris Northwood. Cryptography, attacks and countermeasures @ONLINE.

[34] Axel York Poschmann. *Lightweight cryptography: cryptographic engineering for a pervasive world*. PhD thesis, Ruhr-University Bochum, Germany, 2009.

[35] Tomasz Rachwalik, Janusz Szmidt, Robert Wicik, and Janusz Zablocki. Generation of nonlinear feedback shift registers with special-purpose hardware. In *Communications and Information Systems Conference (MCC), 2012 Military*, pages 1–4. IEEE, 2012.

[36] Juan Soto. Statistical testing of random number generators @ONLINE.

[37] Francois-Xavier Standaert, Gilles Piret, and Jean-Jacques Quisquater. Cryptanalysis of block ciphers: A survey. *UCL Crypto Group Technical Report Series, Technical Report CG-2003*, 2, 2003.

[38] Irfan Syamsuddin, Tharam Dillon, Elizabeth Chang, and Song Han. A survey of rfid authentication protocols based on hash-chain method. In *Convergence and Hybrid Information Technology, 2008. ICCIT'08. Third International Conference on*, volume 2, pages 559–564. IEEE, 2008.

[39] Ton Van Deursen and Sasa Radomirovic. Attacks on rfid protocols. *IACR Cryptology ePrint Archive*, 2008:310, 2008.

[40] Yang Wang, Mark Manulis, Man Ho Au, and Willy Susilo. Relations among privacy notions for signcryption and key invisible sign-then-encrypt. In *Information Security and Privacy*, pages 187–202. Springer, 2013.

[41] Yue Wu, Joseph P Noonan, and Sos Agaian. A novel information entropy based randomness test for image encryption. In *Systems, Man, and Cybernetics (SMC), 2011 IEEE International Conference on*, pages 2676–2680. IEEE, 2011.

[42] Kencheng Zeng, C-H Yang, D-Y Wei, and TRN Rao. Pseudorandom bit generators in stream-cipher cryptography. *Computer*, 24(2):8–17, 1991.

[43] Yuliang Zheng. Digital signcryption or how to achieve cost (signature & encryption) cost (signature)+ cost (encryption). In *Advances in Cryptology-CRYPTO'97*, pages 165–179. Springer, 1997.

[44] Mohd Zaid Waqiyuddin Mohd Zulkifli. Attack on cryptography @ONLINE, April 2008.