IMPLEMENTATION AND COMPARISON OF THE ADVANCED ENCRYPTION STANDARD FINALIST ALGORITHMS ON TMS320C54X

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY
AHMET VOLKAN SERTER

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

DECEMBER 2005

Prof. Dr. Yalçın Tanık	(METU, EE)_______________
Assoc. Prof. Dr. Melek D. Yücel	(METU, EE)_______________
Prof. Dr. Rüyal Ergül	(METU, EE)_______________
Prof. Dr. Buyurman Baykal	(METU, EE)_______________
Özgür Güleryüz	(ASELSAN)_______________

ABSTRACT

IMPLEMENTATION AND COMPARISON OF THE ADVANCED ENCRYPTION STANDARD FINALIST ALGORITHMS ON TMS320C54X

Serter, Ahmet Volkan

M.Sc., Department of Electrical and Electronics Engineering

Supervisor: Assoc. Prof. Dr. Melek D. Yücel

December 2005, 77 pages

Implementation aspects of Advanced Encryption Standard (AES) Contest finalist algorithms (MARS, RC6, RIJNDAEL, SERPENT and TWOFISH) are studied on TMS320C54X processor. The C codes written by Brian Gladman in 1999 are adapted to TMS320C54X and the speed and memory usage values are compared with the adaptation of Karol Gorski and Michal Skalski's implementation in 1999. The effects of implementation environment are investigated by comparing the two implementations. The sensitivities of the finalist algorithms to plaintext, key and key length variations together with the possible reasons are studied and scrutinized.

Three of the algorithms, MARS, RC6 and RIJNDAEL, are implemented on the same platform by using the assembler language. The results show that assembler implementations are improved with respect to C implementations 13% for MARS, 16-20% for RIJNDAEL and 21-28% for RC6.

Keywords: AES, Assembler Implementation, Implementation Aspects, TMS320C5416, Key, Plaintext and Key Length Variation

ÖZ

GELİŞKİN ŞİFRELEME STANDARD YARIŞMASI FİNALİSTİ KRİPTO ALGORİTMALARININ TMS320C54X ÜZERİNDE GERÇEKLEŞTİRİLMESİ VE KARŞILAŞTIRILMASI

Serter, Ahmet Volkan

Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü

Tez yöneticisi: Doç. Dr. Melek D. Yücel

Aralık 2005, 77 sayfa

Gelişkin Şifreleme Standardı yarışmasındaki finalist kripto algoritmalarının (MARS, RC6, RIJNDAEL, SERPENT and TWOFISH) TMS320C54X işlemcisi üzerindeki gerçekleştirilme özellikleri incelenmiştir. Brian Gladman tarafından 1999'da yazılan C kodları TMS320C54X platformuna uyarlanarak, algoritmaların hafıza kullanım ve hız değerleri Karol Gorski and Michal Skalski tarafından 1999'da aynı platformda yapılan uyarlamayla karşılaştırılmıştır. Algoritmaların açık metin, anahtar ve anahtar uzunluğu değişimlerine karşı hassasiyetleri olası nedenleriyle birlikte açıklanmış ve irdelenmiştir. Algoritmalardan üçü, MARS, RC6 ve RIJNDAEL, aynı platform üzerinde makine diliyle gerçeklenmiştir. Elde edilen sonuçlar

MARS için %13, RIJNDAEL için %16-20 ve RC6 için %21-28 arası iyileşme olduğunu göstermektedir.

Anahtar Kelimeler: AES, Makine Dili Gerçeklenmesi, Uygulama Özellikleri, TMS320C5416, Yapısal Karşılaştırma, Anahtar, Açık Metin ve Anahtar Uzunluğu Değişimi

To My Family

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to my advisor, Assoc. Prof. Dr. Melek D. Yücel for her encouragement and support in every stage of this research.

I would like to give special thanks to my home-mates and colleagues for their encouragement and support.

Finally, I would like to express my deep gratitude to all who have encouraged and helped me at the different stages of this work.

And my family, I thank them for everything.

TABLE OF CONTENTS

APPENDICES

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In today's world, information security plays a major role in our life. Millions of people use electronic mail, electronic banking, medical databases, and electronic commerce requiring the exchange of private information. Information security is also the main concern of military applications where the important data, which must not be revealed by enemy, is kept and transferred between military units. To achieve secure communication and storage of information in both military and civil fields, cryptographic techniques are used. There are two types of cryptographic techniques as public key cryptography and symmetric key cryptography.

Public key cryptography has two types of keys called public key and private key, which are not symmetric, belonging to same person. The data to be sent is encrypted by the public key of the receiver side at the sender side and decrypted by the private key of the receiver side. The public key as its name implies can be reached by everyone, but one cannot obtain the private key from the public key.

Symmetric key cryptography uses the same key for encrypting and decrypting operations on the sender and receiver sides. For this reason key must be kept secret, so it is also called as secret key cryptography. There are two types of symmetric key algorithms: stream ciphers and block ciphers.

Block ciphers encrypt the plaintext in blocks by taking the plaintext in fixed-length blocks as input and give the ciphertext again in fixed-length blocks as output. Stream ciphers encrypt the plaintext character-by-character (or bit by bit). These types of algorithms produce a stream of bits, which is called the key stream of the algorithm. This key stream is used to encrypt or decrypt the plaintext or the ciphertext.

In a typical session, which means the exchange of the private data, a public key algorithm is used for the exchange of a session key and to provide authenticity. The session key is then be used in conjunction with a symmetric key algorithm. Symmetric key algorithms tend to be significantly faster than public key algorithms and as a result are typically used in bulk data encryption. The most widely used public key algorithm has become RSA and symmetric key algorithm has become DES (Data Encryption Standard).

DES was declared as a standard for symmetric encryption algorithm by National Bureau of Standards (NBS) in the early 1970s. This standard has persisted for over 30 years until linear [Matsui, 1993] and differential [Biham, Shamir, 1993] attacks showed that it was possible to resolve DES by not using brute force attack. After this development, Triple-DES was created. Triple-DES is a variant of DES, which increases the key length therefore increasing the security with degradation in speed.

Realizing that DES was no longer appropriate for many cryptographic applications, National Institute of Standards and Tables (NIST) initiated the efforts of finding a new algorithm as Advanced Encryption Standard (AES). The search began in 1997. NIST set several criteria for an appropriate algorithm. The algorithm must be a symmetric block cipher capable of handling key sizes of 128, 196 and 256 bits. Secondly, the algorithm must be available worldwide on a royalty free basis. Another property of algorithms was simplicity for understanding and cryptanalysis of the algorithm. Another requirement was that the algorithm would be suitable on many platforms and for many purposes. Successful algorithms were also to demonstrate resistance to both linear and differential cryptanalysis.

There were 21 submissions for the standard, 15 of which met NIST's basic criteria, in Round 1 evaluation. These were CAST-256, CRYPTON, DEAL, DFC, LOKI97, E2, FROG, HASTY PUDDING CIPHER, MAGENTA, MARS, RC6, RIJNDAEL, SAFER+, SERPENT and TWOFISH. After examination of these 15 algorithms, NIST announced the five finalists to be MARS [Burwick, 1999], RC6 [Rivest, 1998], Rijndael [Daemen, Rijmen, 1999], Serpent [Anderson, Biham, Knudsen, 1998] and Twofish [Schneier, 1998] for a further evaluation, Round 2, in August of 1999.

After examining the round 2 algorithms in terms of security, cost, and implementation characteristics, NIST announced that RIJNDAEL was selected as the proposed AES on October 2, 2000.

In this study, implementation aspects of Round 2 algorithms are studied. The C codes written by Brian Gladman [Gladman, 1999] are adapted to the TMS320C5416 DSK development environment. The implementation realized by Karol Gorski and Michal Skalski [Gorski, Skalski, 1999], which was used for Round 1 evaluation, is examined and evaluated. The effects of implementation environment are investigated by comparing Karol Gorski and Michal Skalski's [Gorski, Skalski, 1999] implementation with ours. The sensitivities of the finalist algorithms to plaintext, key and key length variations together with the possible reasons leading sensitivities are studied and the significances of these sensitivities in the sense of implementation are scrutinized. Three of algorithms, MARS, RC6, RIJNDAEL, are implemented on the same platform by using assembler. Finally, the results of assembler implementation are compared with the C implementation.

In Chapter 2, building blocks of encryption, decryption and key setup operations of AES finalist are described.

In Chapter 3, the effects of development environment over implementation properties and algorithms' sensitivities to the plaintext, key and key length changes together are studied.

In Chapter 4, assembler implementations of MARS, RC6 and RIJNDAEL are explained. The results of assembler implementations are compared with the C implementation.

In Chapter 5, concluding remarks are discussed.

CHAPTER 2

DESCRIPTION OF AES FINALIST ALGORITHMS

In this chapter, after examining the structures and building blocks of AES Finalist Algorithms in the sense of implementation, a comparison of algorithms are given in terms of building blocks and operational properties.

2.1 DESCRIPTION OF ALGORITHMS

AES Finalist Algorithms are examined by dividing an algorithm into three parts as encryption, decryption, and key setup. We examine encryption operation for each algorithm by giving a brief sketch and indicate the differences of decryption operation with respect to encryption. After examining encryption and decryption, key setup operation is examined in the following sections.

2.1.1 MARS

MARS [Burwick, 1999] encryption operation has key addition block as pre-whitening at the beginning and then 8 rounds of unkeyed forward mixing block, then eight rounds of keyed forward transformation block, then 8 rounds of keyed backwards transformation block, then eight rounds of unkeyed backwards mixing block, and finally key subtraction block as post-whitening. MARS processes the data in 32-bit blocks. The block diagram summarizing the structure of encryption is in Figure 2.1

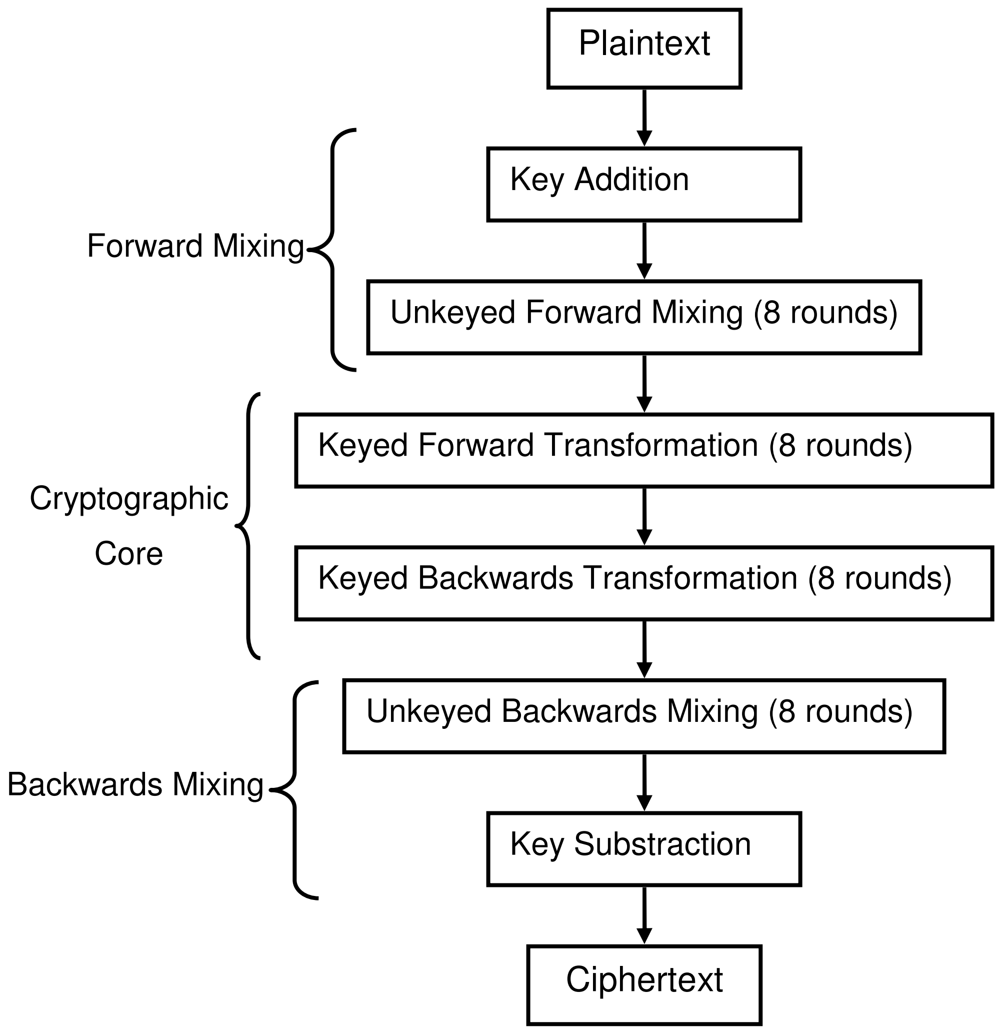

Plaintext
Key Addition
Forward Mixing
Unkeyed Forward Mixing (8 rounds)
Keyed Forward Transformation (8 rounds)
Cryptographic Core
Keyed Backwards Transformation (8 rounds)
Unkeyed Backwards Mixing (8 rounds)
Backwards Mixing
Key Substraction
Ciphertext

Figure 2.1. MARS Block Diagram

The key addition together with the unkeyed forward mixing is called forward mixing, which uses two 8x32 bit S-boxes, 32-bit addition, and the XOR operation. The 8 forward and 8 backwards keyed transformations are called the cryptographic core, which uses 32-bit key multiplication, data-dependent rotations, key addition, two 8x32 bit S-boxes, 32-bit addition, and XOR operation. Backwards mixing consists of unkeyed backwards mixing and key subtraction utilizing two 8x32 bit S-boxes, 32-bit addition, and the

XOR operation like the forward mixing. Both the mixing and the core rounds are modified Feistel [Feistel, 1973] rounds in which one fourth of the data block is used to alter the other three fourths of the data block. The decryption process is the inverse of the encryption process. It uses the same operations as encryption by combining them in a reverse way. The key setup operation can take key values starting from 128-bit to 448-bit and generates forty 32-bit expanded key by using XOR, 32-bit addition, a 9x32 bit S-box, which is a combination of two 8x32 bit S-boxes and data dependent rotation operations.

2.1.2 RC6

RC6 [Rivest, 1998], encryption consists of key addition at the beginning and end as pre- and post whitening respectively and a round function in between. It processes the plaintext in 32-bit words. The structure of RC6 encryption can be summarized in Figure 2.2.

Key addition is used for pre- and post-whitening having 32-bit addition. The round function of RC6 uses data dependent rotations combined with a quadratic function of the data. Each round includes 32-bit multiplication, addition, XOR, and key addition operations. RC6's round function is a Feistel [Feistel, 1973] structure. RC6's round function is the only one among the finalists that does not use any S-box. Decryption is the reverse operation of encryption and it uses the same operations like encryption. The key setup operation can take 128 bit, 192 bit, and 256 bit long keys and returns forty-four 32-bit subkeys using 32-bit addition and data dependent rotation.

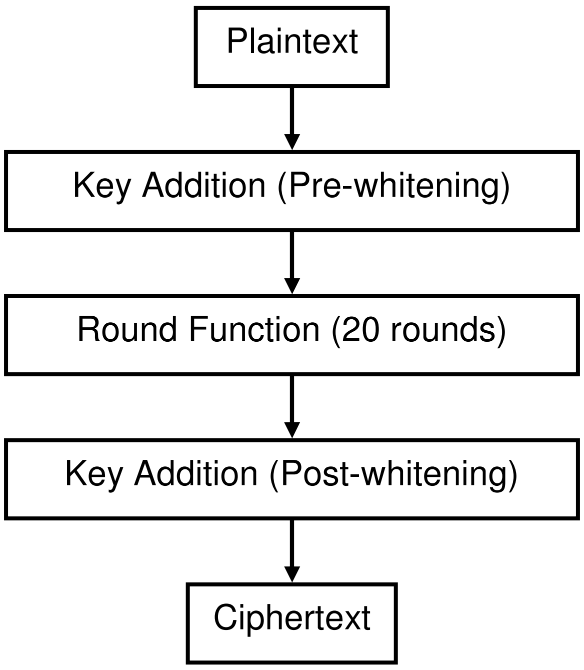

Plaintext
Key Addition (Pre-whitening)
Round Function (20 rounds)
Key Addition (Post-whitening)
Ciphertext

Figure 2.2. RC6 Block Diagram

2.1.3 RIJNDAEL

RIJNDAEL's [Daemen, Rijmen, 1999] encryption consists of the key addition block and a round function, which is iterated 10, 12 or 14 times depending on the key size, see Figure 2.3.

Rijndael processes data by partitioning it into a two dimensional array of bytes. RIJNDAEL's round function consists of four steps. In the first step, an 8x8 S-box, which can be realized by a table look-up, is applied to each byte. The second step shifts the rows of the array and the third step mixes the columns of the array. Therefore, they are called linear mixing steps. In the fourth step, subkey bytes are XORed into each byte of the array, which can be defined as GF (2) addition. In the last round, the column mixing is omitted. The decryption is the inverse of the encryption and it uses the same building blocks in a reverse structure. The key setup takes 128, 192 or 256 bit long keys and generates subkeys according to number of rounds, which can be 10, 12 or 14, using S-box of encryption and XOR operation.

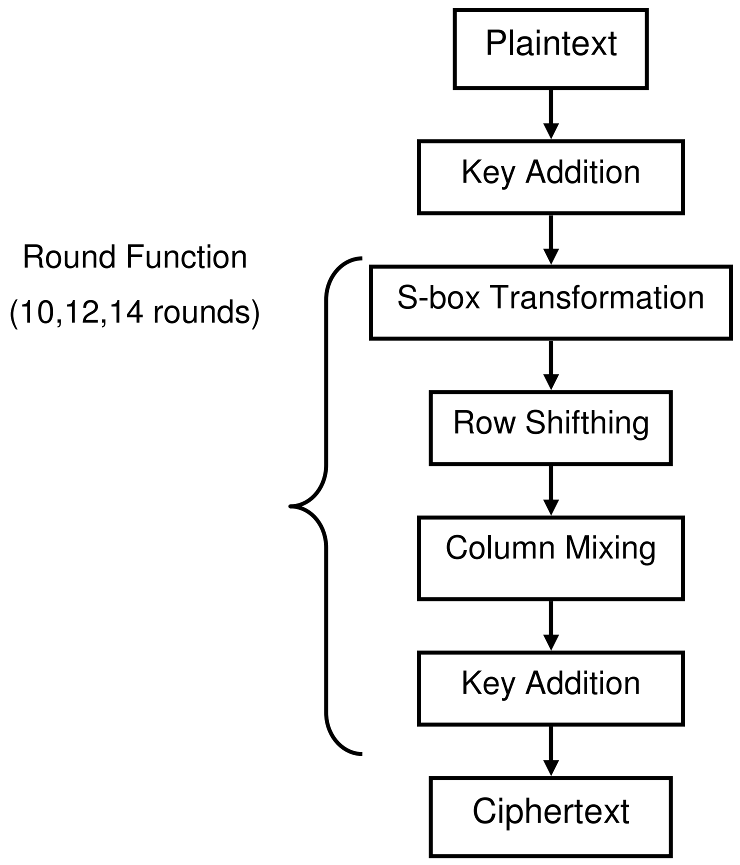


Figure 2.3. RIJNDAEL Block Diagram

2.1.4 SERPENT

Serpent [Anderson, Biham, Knudsen, 1998] is a substitution-linear transformation network consisting of 32 rounds, which is different from Feistel [Feistel, 1973] structure. Figure 2.4 shows three main building blocks of encryption. The first and the last blocks are non-cryptographic initial and final permutations that facilitate an alternative mode of implementation called the bitslice mode. The second block is the round function.

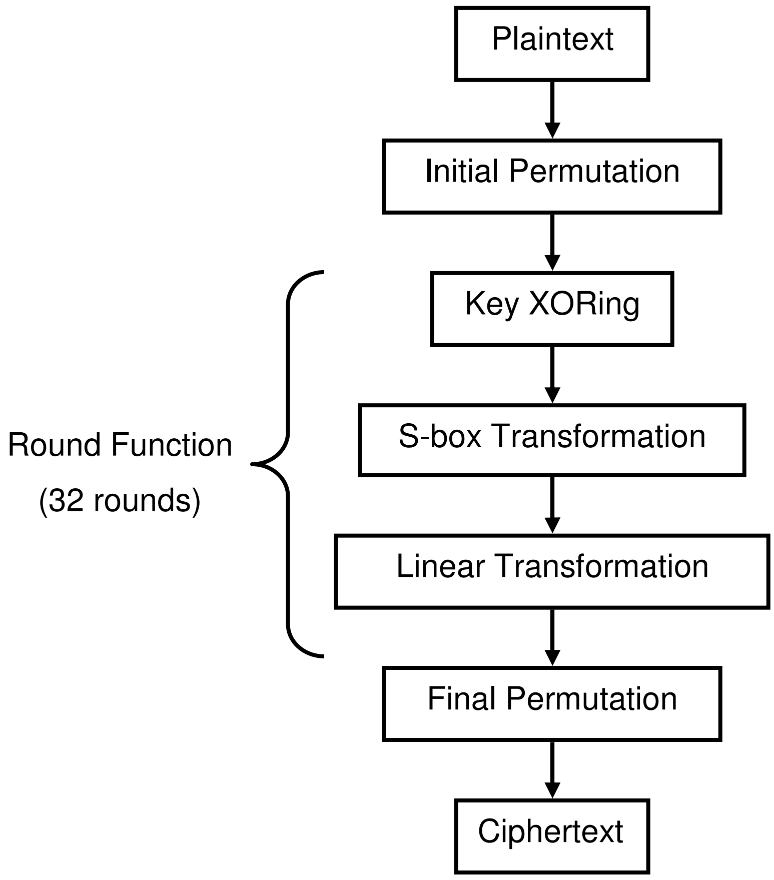
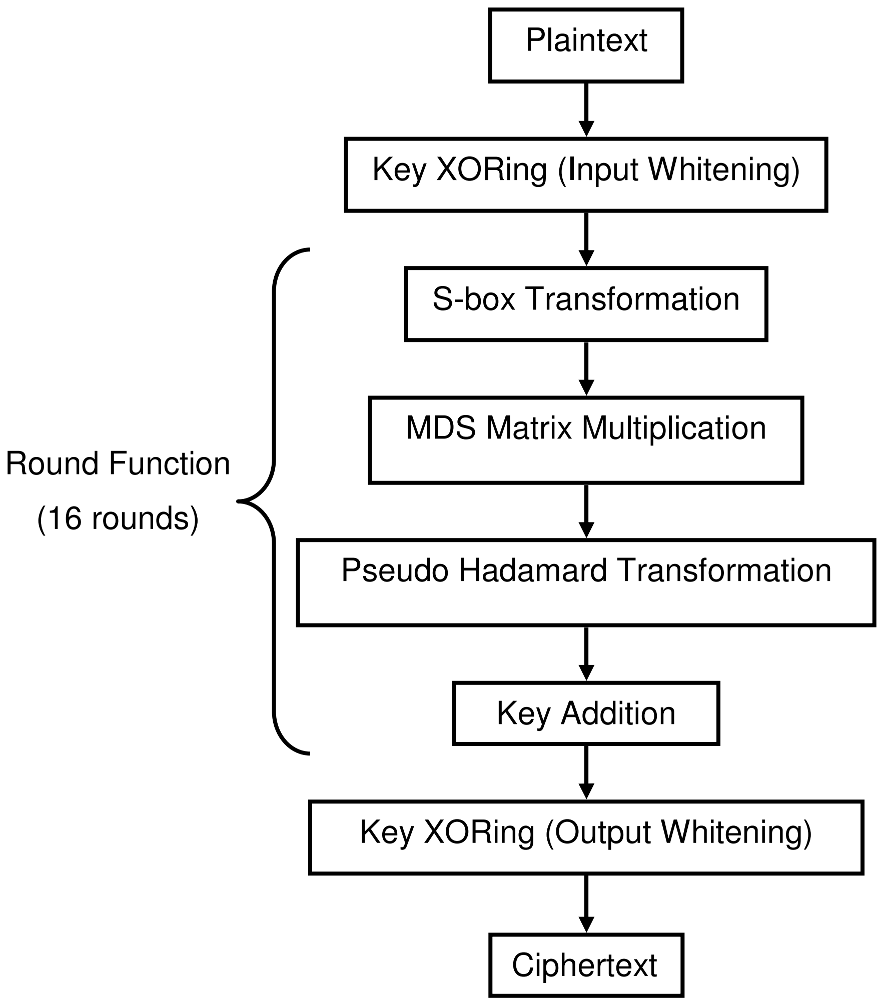


Figure 2.4. SERPENT Block Diagram

The round function consists of three layers: the key XOR operation, 32 parallel applications of one of the eight specified 4x4 S-boxes, and a linear transformation. In the last round, a second layer of key XOR replaces the linear transformation. Decryption operation is different from encryption. It is realized by using the inverse of the S-boxes in the reverse order and the inverse linear transformation and the subkeys in reverse order. The key setup takes 128, 192 or 256 bit long key and generates 132 32-bit long subkeys by using eight S-boxes, an affine transformation and the initial permutation used in the cipher.

2.1.5 TWOFISH

Encryption consists of input whitening, a round function, which is a slightly modified Feistel [Feistel, 1973] network using 1-bit rotations, iterated 16 times and output whitening [Schneier, 1998], see Figure 2.5.



Plaintext
Key XORing (Input Whitening)
S-box Transformation
MDS Matrix Multiplication
Round Function (16 rounds)
Pseudo Hadamard Transformation
Key Addition
Key XORing (Output Whitening)
Ciphertext

Figure 2.5. TWOFISH Block Diagram

The round function acts on 32-bit words with four key dependent 8x8 S-boxes, followed by a fixed 4x4 maximum distance separable matrix over GF(2^8), a pseudo-hadamard transform (PHT), and key addition. The decryption uses the same blocks as encryptions, which are combined in a reverse manner. The key setup takes 128, 192 or 256-bit long keys and generates 40 32-bit long subkeys by making use of MDS matrix, 32-bit addition, PHT and XOR operation. It also outputs key dependent S-boxes, which are used for encryption and decryption.

2.2. STRUCTURAL COMPARISON OF AES FINALIST ALGORITHMS

Structural comparison of AES finalist algorithms is given together with the operational comparison after explaining the Feistel Structure, [Feistel, 1973], which is used in some of finalists.

2.2.1 Feistel Structure

The Feistel Structure [Feistel, 1973] is a method used in many encryption algorithms because of its well defined architecture. Three of the AES finalist algorithms (MARS, RC6 and TWOFISH) use this structure. The structure consists of N rounds, which are equivalent, and a final swap. In each round, round input is divided into two parts as left and right, and then the right part is passed as the left part of next round. The right part is inputted to an irreversible function together with the round key and is set as the right part of the next round by XORing with the left part. In the final swap part, only right and left parts are swapped. Figure 2.6 explains the Feistel Structure.

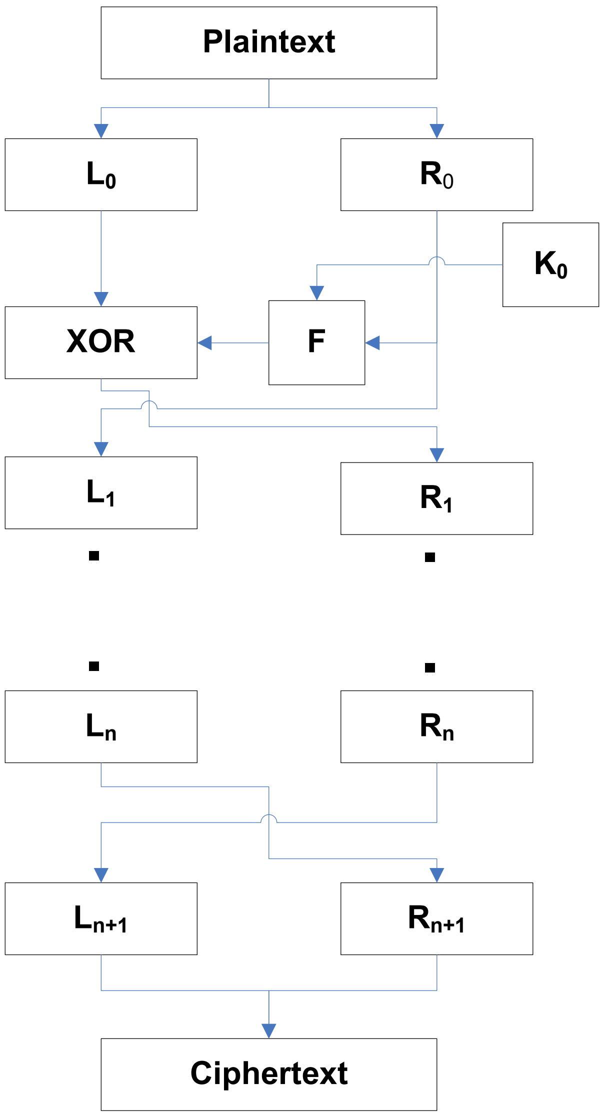


Figure 2.6. Feistel Structure

2.2.2 Structural Comparison

Table 2.1 shows the structural differences of algorithms. The Feistel Structure is used by MARS, RC6 and TWOFISH. Round number is fixed for all algorithms except RIJNDAEL and it varies between 10 and 14 according to key length.RC6 is the only one not using any S-box. S-boxes are also used in key scheduling for RIJNDAEL, SERPENT and TWOFISH. S-box sizes of RIJNDAEL and TWOFISH are the same. All of the algorithms can work with key sizes of 128, 192 and 256. MARS' can also work with the any key length varying between 128 and 448. SERPENT is the only algorithm that does not use any initial or final key addition.

Table 2.1. Structural Comparison of AES Finalist Algorithms

	MARS	**RC6**	**RIJNDAEL**	**SERPENT**	**TWOFISH**
Feistel Structure	Used	Used	Not Used	Not Used	Used
Number of Rounds	16	20	10,12, 14[2]	32	16
Number of S-Boxes	2	no S-Box	single	8	4
S-Box Size	8x32	-	8x8	4x4	8x8
Key Length	Between128 and 448[3] bits	128,196, 256 bits	128,196, 256 bits	128,196, 256 bits	128,196, 256 bits
Initial Key Addition	Used	Used	Used	Not Used	Used
Final Key Addition/ Subtraction	Subtraction Used	Addition Used	Not Used	Not Used	Addition Used
S-Box Usage in Key Scheduling	Not Used	Not Used	Used	Used	Used

A comparison of algorithms in terms of operations used can be seen in Table 2.2. Table 2.2 shows that all of the algorithms use XOR and fixed rotation operations. GF (2^8) multiplication is utilized by RIJNDAEL and TWOFISH. Mod 2^{32} multiplication is used by MARS and RC6. Mod 2^{32} addition and data dependent rotations are used by MARS, RC6 and TWOFISH.

Table 2.2. Operational Comparison of AES Finalist Algorithms

	MARS	**RC6**	**RIJNDAEL**	**SERPENT**	**TWOFISH**
Xor Operation	Used	Used	Used	Used	Used
GF(2^8) multiplication	Not Used	Not Used	Used	Not Used	Used
mod 2^{32} multiplication	Used	Used	Not Used	Not Used	Not Used
mod 2^{32} addition	Used	Used	Not Used	Not Used	Used
Data Dependent rotations	Used	Used	Not Used	Not Used	Used
Fixed rotations/ shifts	Used	Used	Used	Used	Used

CHAPTER 3

PERFORMANCE ANALYSIS OF AES FINALIST ALGORITHMS IMPLEMENTED IN C LANGUAGE

In this chapter, performance figures of AES finalist algorithms, which were obtained by Karol Gorski and Michal Skalski [Gorski, Skalski, 1999] are presented. We repeat the same tests as Karol Gorski and Michal Skalski using a similar processor but a different environment and compare our results with the previous work in section 3.1. The sensitivity of the speed performance to plaintext, key and key length changes is investigated in sections 3.2, 3.3, and 3.4 respectively.

3.1 COMPARISON OF AES FINALIST ALGORITHMS ACCORDING TO DEVELOPMENT ENVIRONMENT

In this section, after the presentation of previous work by Karol Gorski and Michal Skalski [Gorski, Skalski, 1999], we give our test results in section 3.1.2 and finally the comparison in section 3.1.3.

3.1.1 Analysis Made by Karol Gorski and Michal Skalski

Performance comparison of AES Finalist Algorithms on Texas Instruments' digital signal processor, TMS320C541, was made by Karol Gorski and Michal Skalski and was presented to NIST for evaluating the finalist algorithms for the first round of the AES contest [Gorski, Skalski, 1999].

Karol Gorski and Michal Skalski used the C codes written by Brian Gladman [Gladman, 1999] and adapted these codes for Texas Instruments' TMS320C541 digital signal processor. When adapting, no effort was made to optimize any of these implementations for the processor. The source codes were compiled using the standard C compiler by TI having version 1.20 with optimization options turned on.

Their comments and recommendations were based on results of timing and memory usage values which were obtained from a real processor residing on a TMS320C54X evaluation module, having a speed of 40 MIPS (millions of instructions per second), using the TI Debugger v1.30 (used as a profiler) and the linker outputs respectively.

3.1.1.1 Test Results

Tests on algorithms were performed on 100 sets of random blocks of plaintext and key. The arithmetic means of cycle counts were calculated and memory usages were taken from the linker output and were expressed in 16-bit words.

The cycle count ranking was created using the minimum of the encryption, decryption, and key setup operations' cycle counts for each algorithm. The memory usage ranking was created using the total memory usages. In order to obtain the speed value for each algorithm, the value 128*4*10^7 was divided by the minimum of encryption, decryption and key setup speeds for that algorithm.

Table 3.1 was obtained the for the cycle counts of key setup, encryption and decryption operations.

Table 3.1. Cycle Counts for Operations of AES Finalists by Karol Gorski and Michal Skalski

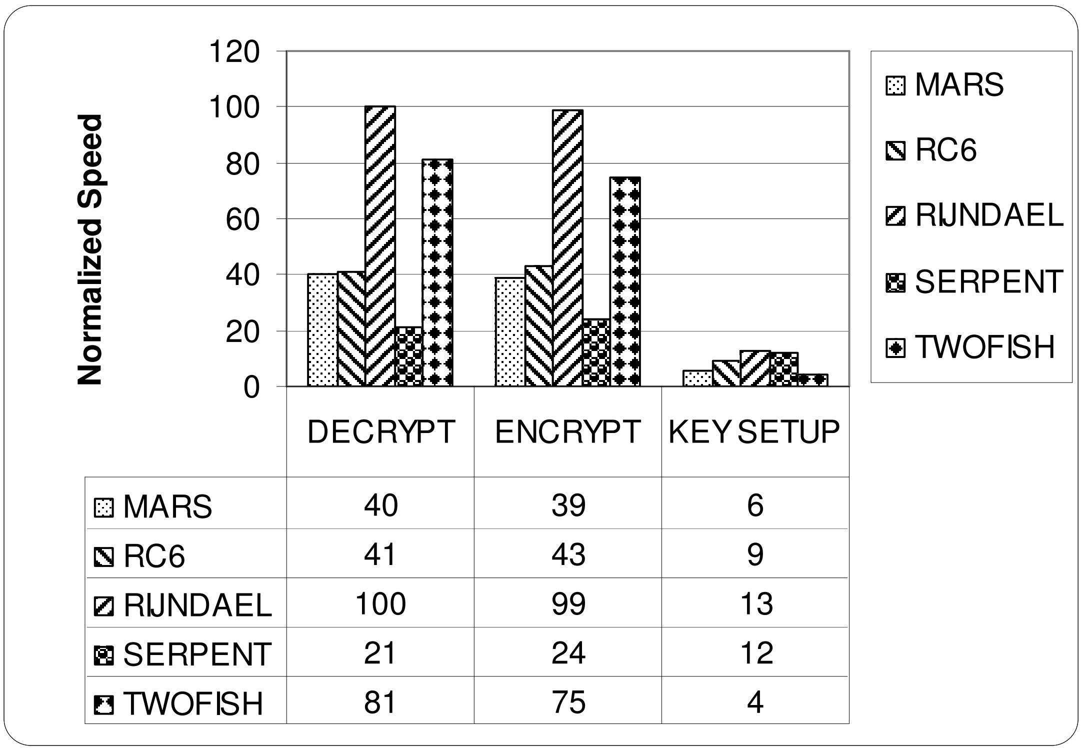
	CYCLE COUNTS			in Mbps	RANK
ALGORITHM	***DECRYPT***	***ENCRYPT***	***KEY SETUP***	***SPEED***	
MARS	8826	8908	54427	0.580	4
RC6	8487	8231	40011	0.622	3
RIJNDAEL	3500	3518	26642	1,463	1
SERPENT	16443	14703	28913	0.348	5
TWOFISH	4328	4672	88751	1,182	2

The speed values of algorithms in terms of mega bits per second are shown in Table 3.2 when using a fixed key throughout the encryption or decryption.

Table 3.2. Speed in Mega Bits per Second for Encryption and Decryption Operations of AES Finalists by Karol Gorski and Michal Skalski

	SPEED in Mbps	
ALGORITHM	***DECRYPT***	***ENCRYPT***
MARS	0.580	0.575
RC6	0.603	0.622
RIJNDAEL	1.463	1.498
SERPENT	0.311	0.348
TWOFISH	1.182	1.096

When the cycle counts shown in Table 3.1 are normalized according to the minimum cycle count value of 3500, which is the decryption cycle count value for Rijndael and multiplied by 100, Figure 3.1 is obtained.



	DECRYPT	ENCRYPT	KEY SETUP
MARS	40	39	6
RC6	41	43	9
RIJNDAEL	100	99	13
SERPENT	21	24	12
TWOFISH	81	75	4

Figure 3.1. Relative Percentages for Speed Performance of AES Finalists by Karol Gorski and Michal Skalski

Figure 3.1 shows that RIJNDAEL has the best speed values for encryption, decryption and key setup operations. TWOFISH has the second rank in both encryption and decryption, but its key setup is the slowest. RIJNDAEL and TWOFISH can be put into the same category especially due to their high encryption and decryption speeds. Key setup of TWOFISH is the worst one among finalists so it can degrade the performance of algorithm when key is changed very frequently. MARS and RC6 have very close speed values, which are slightly higher for RC6, for all operations and can be classified in a second category. Their speed values can not exceed 50% of TWOFISH's speed values for encryption and decryption whereas their key setup speeds are better. SERPENT's encryption and decryption speeds are the worst but its key setup speed is very high which increases the processing speed in applications where the key is changed frequently.

Karol Gorski and Michal Skalski obtained Table 3.3 for the memory usage separately for program, data, and total memory usages.

Table 3.3. Memory Usage (in 16-bit words) of AES Finalist Algorithms by Karol Gorski and Michal Skalski

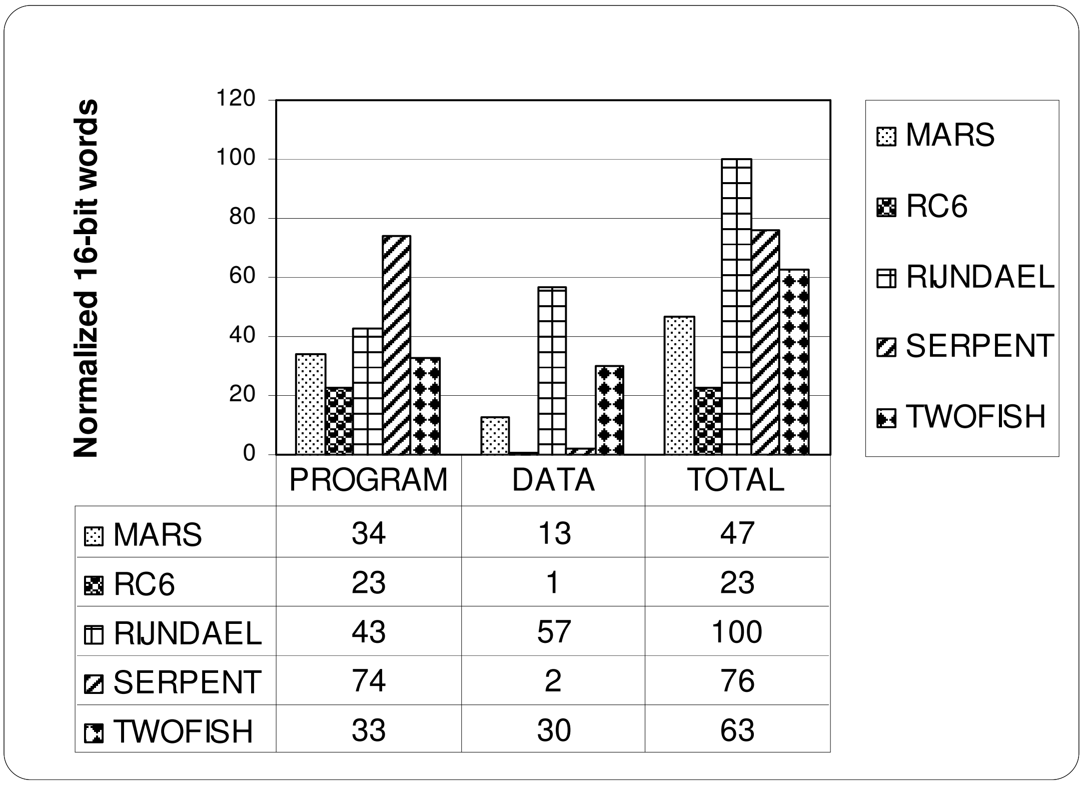
ALGORITHM	***PROGRAM***	***DATA***	***TOTAL***	**RANK**
MARS	5663	2258	7921	2
RC6	3772	104	3876	1
RIJNDAEL	7221	9510	16731	5
SERPENT	12373	304	12677	4
TWOFISH	5590	4950	10540	3

The memory usage values in Table 3.3 can be expressed in terms of kilo words in Table 3.4.

Table 3.4. Memory Usage (in Kilo Words) of AES Finalist Algorithms by Karol Gorski and Michal Skalski

ALGORITHM	***PROGRAM***	***DATA***	***TOTAL***
MARS	5.5K	2.2K	7.7K
RC6	3.7K	0.1K	3.8K
RIJNDAEL	7.1K	9.3K	16.4K
SERPENT	12.1K	0.3K	12.4K
TWOFISH	5.5K	4.8K	10.3K

When the memory usage values shown in Table 3.3 are normalized according to the maximum memory usage value of 16731, which is the value for the total memory usage for Rijndael and multiplied by 100, Figure 3.2 is obtained.



	PROGRAM	DATA	TOTAL
MARS	34	13	47
RC6	23	1	23
RIJNDAEL	43	57	100
SERPENT	74	2	76
TWOFISH	33	30	63

Figure 3.2. Relative Percentages for Memory Usage Performances of AES Finalists by Karol Gorski and Michal Skalski

Figure 3.2 shows that program memory usage of SERPENT is the maximum among all algorithms whereas its data memory usage is very low which is very suitable for platforms not having an external memory. The remaining algorithms have close values of program memory usage, which are nearly the half of SERPENT. RC6's data memory usage is also very low with respect to MARS, RIJNDAEL and TWOFISH, which is an advantage for low external memory requiring applications. Although the algorithms have variable memory usage values, they are all in the range of the program and data memory capacity of TMS320C541 (10 K for data and 56 K for program memory) which is a moderate value for modern processors. Therefore, memory usage values do not have any influence on the implementation of algorithms in this platform.

3.1.2 Analysis of AES Finalist Algorithms TMS320C5416 DSK Stand-Alone Development and Evaluation Module

We investigate the performances of AES finalist algorithms in terms of speed and memory usage by using 100 random plaintexts and random keys. Objective of this investigation is to compare the algorithms in a similar processor but a development environment different from Gorski and Skalski's development environment.

3.1.2.1 Description of Development Environment

The tests are performed on TMS320C5416 DSK stand-alone development and evaluation module. The key features of the TMS320C5416 DSK can be summarized as follows:

- VC5416 operating at 16-160 MHz
- On board USB JTAG controller with plug and play drivers
- 64K words of on board RAM
- 256K words of on board Flash ROM
- Expansion Connectors (Memory Interface, Peripheral Interface, and Host Port Interface)
- On board IEEE 1149.1 JTAG Connection for Optional Emulation Debug
- Burr Brown PCM 3002 Stereo Codec
- +5 volt operation

3.1.2.2 Random Plaintext and Key Generation

In this method, output of each encryption is fed as the plaintext to the next encryption. The key of each encryption is made up of the ciphertext two entries back, except for the first and second encryptions. After each encryption is completed, a decryption is applied. By using such a method, randomness of keys and plaintexts are guaranteed since output of each encryption unit is truly random. See Figure 3.3 for the first three encryption /decryption pairs. We start with the plaintext “11223344112233441122 334411223344” and all zero key in our implementation.

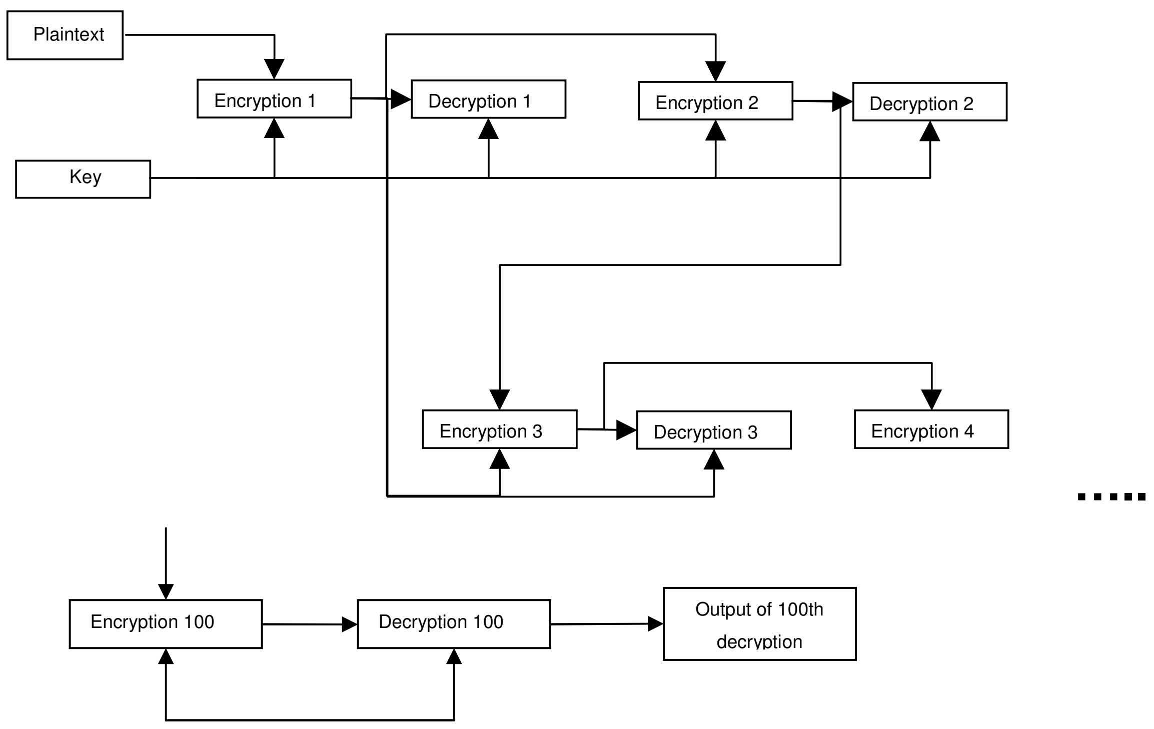


Figure 3.3. Random Plaintext and Key Generation

3.1.2.3 Test Results

Table 3.5 is obtained for the cycle counts of key setup, encryption and decryption operations.

Table 3.5. Cycle Counts for Operations of AES Finalists

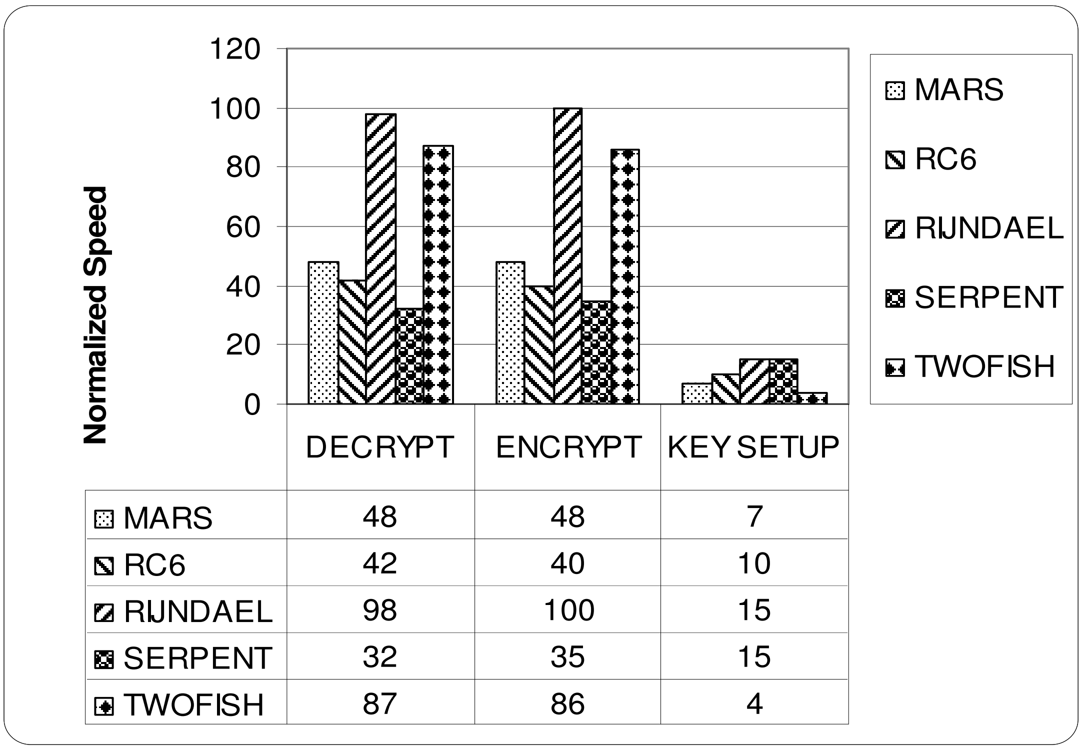
	CYCLE COUNTS			in Mbps	
ALGORITHM	***DECRYPT***	***ENCRYPT***	***KEY SETUP***	***SPEED***	**RANK**
MARS	4082	4110	28220	1.25	3
RC6	4694	4889	20595	1.09	4
RIJNDAEL	2017	1970	13253	2.59	1
SERPENT	6174	5681	13512	0.9	5
TWOFISH	2252	2296	52708	2.27	2

The speed values of algorithms in terms of mega bits per second are shown in Table 3.6 when using a fixed key throughout the encryption or decryption.

Table 3.6. Speed Values in Mbps for Operations of AES Finalists

	SPEED in Mbps	
ALGORITHM	***DECRYPT***	***ENCRYPT***
MARS	1.25	1.25
RC6	1.09	1.05
RIJNDAEL	2.53	2.59
SERPENT	0.83	0.90
TWOFISH	2.27	2.23

When the cycle counts shown in Table 3.5 are normalized according to the minimum cycle count value of 1970, which is the value for the encryption operation for Rijndael and multiplied by 100, Figure 3.4 is obtained.



	DECRYPT	ENCRYPT	KEY SETUP
MARS	48	48	7
RC6	42	40	10
RIJNDAEL	98	100	15
SERPENT	32	35	15
TWOFISH	87	86	4

Figure 3.4. Relative Percentages for Speed Performance of AES Finalists

Figure 3.4 shows that RIJNDAEL is in the same category, which is the first category, with TWOFISH and MARS forms another category, which is the second category, with RC6 for encryption and decryption operations whereas SERPENT can not be included in none of these categories because of its relatively low speed values as Gorski and Skalski's implementation. The only difference between our implementation and previous implementation in terms of encryption and decryption speed rank is that MARS has better speed values with respect to RC6 in our implementation. Key setup speed rank for our implementation remains the same as the previous one.

Table 3.7 shows the memory usage of algorithms in terms of 16-bit words.

Table 3.7. Memory Usage (in 16-bit Words) of Algorithms

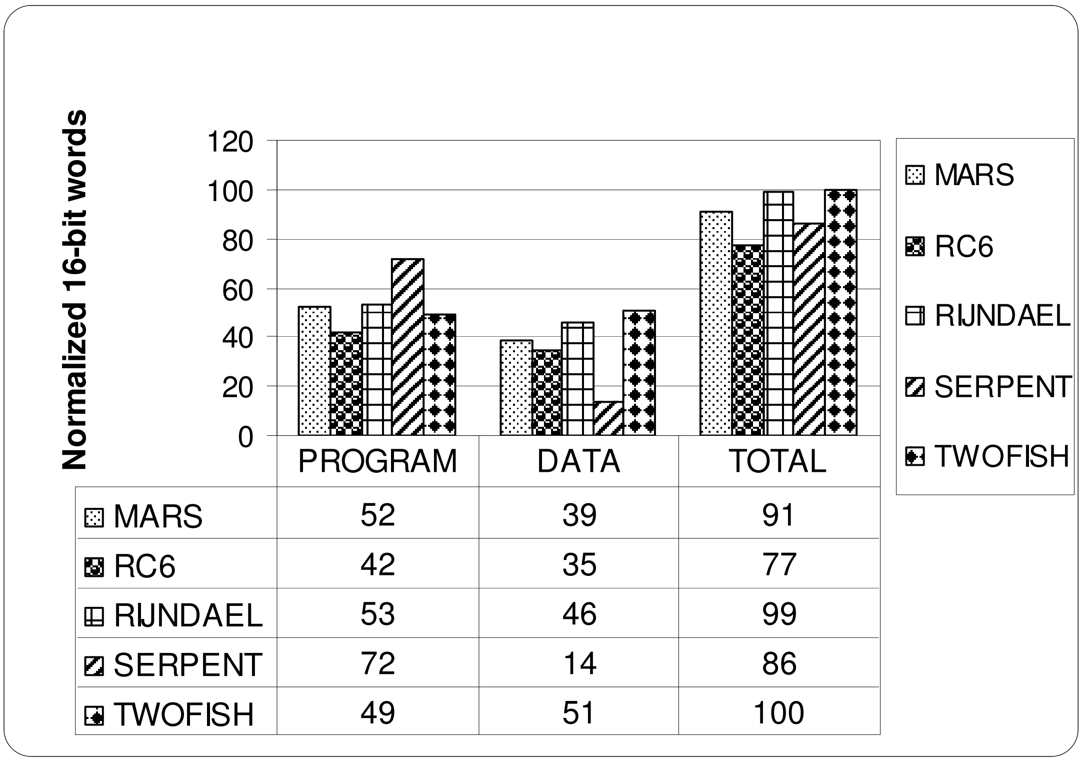
ALGORITHM	***PROGRAM***	***DATA***	***TOTAL***	**RANK**
MARS	15016	11139	26155	3
RC6	12174	10027	22201	1
RIJNDAEL	15297	13301	28598	4
SERPENT	20699	4083	24782	2
TWOFISH	13960	14817	28777	5

When values in Table 3.7 are expressed in terms of kilo words, Table 3.8 is obtained.

Table 3.8. Memory Usage of Algorithms (in Kilowords)

ALGORITHM	***PROGRAM***	***DATA***	***TOTAL***	**RANK**
MARS	14.7K	10.9K	25.6K	3
RC6	11.9K	9.9K	21.8K	1
RIJNDAEL	14.9K	13K	27.9K	4
SERPENT	20.2K	4K	24.2K	2
TWOFISH	13.6K	14.5K	28.1K	5

When the memory usage values shown in Table 3.7 are normalized according to the maximum memory usage value of 28777, which is the value for the total memory usage for Twofish and multiplied by 100, Figure 3.5 is obtained.



	PROGRAM	DATA	TOTAL
MARS	52	39	91
RC6	42	35	77
RIJNDAEL	53	46	99
SERPENT	72	14	86
TWOFISH	49	51	100

Figure 3.5. Relative Percentages of Memory Usage for AES Finalists

Figure 3.5 shows that all of the algorithms except SERPENT can be classified in one category in terms of program memory and data memory usages. SERPENT's program memory usage is high with respect to other algorithms while its data memory usage is relatively very low which helps SERPENT to be implemented in low external data memory requiring platforms.

Table 3.7 shows that memory usage values for all algorithms are in the range of internal memory capacity of TMS320C5416, which are 32K for program memory and 256 K for data memory. Therefore, memory usage values of algorithms do not cause a bottleneck for our implementation.

3.1.3 Comparison of Algorithms According to Development Environments

In this section, each AES Finalist algorithm is investigated in terms of cycle count and memory usage values according to our and previous implementations.

In Table 3.9, cycle count values are given according to implementation. After normalization of the values, relative speed values are given in Figure 3.6.

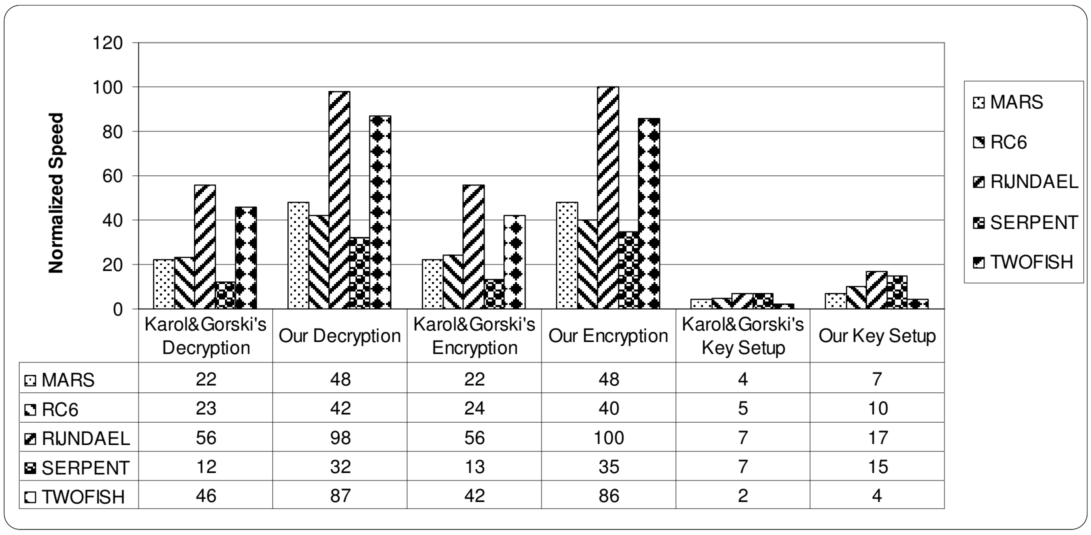
Table 3.9. Cycle Count Values According to Implementation

	DECRYPTION		**ENCRYPTION**		**KEY SETUP**	
	Karol&Gorski's	***Ours***	***Karol&Gorski's***	***Ours***	***Karol&Gorski's***	***Ours***
MARS	8826	4082	8908	4110	54427	28220
RC6	8487	4694	8231	4889	40011	20595
RIJNDAEL	3500	2017	3518	1970	26642	11464
SERPENT	16443	6174	14703	5681	28913	13512
TWOFISH	4328	2252	4672	2296	88751	52708

Table 3.10 shows the speed values of encryption and decryption in terms of mbps when a fixed key is used.

Table 3.10. Speed Values in Mbps According to Implementation

	DECRYPTION		**ENCRYPTION**	
	Karol&Gorski's	***Ours***	***Karol&Gorski's***	***Ours***
MARS	0.580	1.25	0.575	1.25
RC6	0.603	1.09	0.622	1.05
RIJNDAEL	1.463	2.53	1.498	2.59
SERPENT	0.311	0.83	0.348	0.90
TWOFISH	1.182	2.27	1.096	2.23



	Karol&Gorski's Decryption	Our Decryption	Karol&Gorski's Encryption	Our Encryption	Karol&Gorski's Key Setup	Our Key Setup
MARS	22	48	22	48	4	7
RC6	23	42	24	40	5	10
RIJNDAEL	56	98	56	100	7	17
SERPENT	12	32	13	35	7	15
TWOFISH	46	87	42	86	2	4

Figure 3.6. Relative Percentages of Speed According to Implementation

Figure 3.6 shows that speeds of encryption and decryption are doubled for MARS, RC6, RIJNDAEL and TWOFISH whereas they are 2.5 times of the previous one for SERPENT. The key setup speeds are doubled for MARS, RC6, RIJNDAEL and SERPENT but it is 30% more than the previous one for TWOFISH.

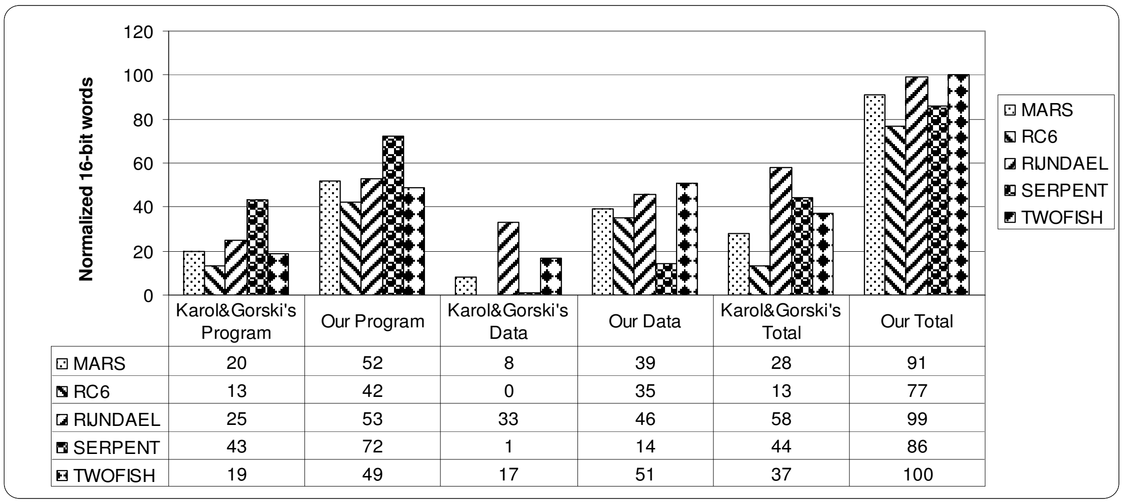
Table 3.11 shows the memory usage values in terms of 16 bit words and Table 3.12 shows the memory usage values in terms of 16 bit kilo words.

Table 3.11. Memory Usage Values According to Implementation (in 16-bit words)

	PROGRAM		**DATA**		**TOTAL**	
	Karol&Gorski's	***Ours***	***Karol&Gorski's***	***Ours***	***Karol&Gorski's***	***Ours***
MARS	5663	15016	2258	11139	7921	26155
RC6	3772	12174	104	10027	3876	22201
RIJNDAEL	7221	15297	9510	13301	16731	28598
SERPENT	12373	20699	304	4083	12677	24782
TWOFISH	5590	13960	4950	14817	10540	28777

Table 3.12. Memory Usage Values (In Kilowords) According to Implementation

	PROGRAM		**DATA**		**TOTAL**	
	Karol&Gorski's	***Ours***	***Karol&Gorski's***	***Ours***	***Karol&Gorski's***	***Ours***
MARS	5.5K	14.7K	2.2K	10.9K	7.7K	25.6K
RC6	3.7K	11.9K	0.1K	9.9K	3.8K	21.8K
RIJNDAEL	7.1K	14.9K	9.3K	13K	16.4K	27.9K
SERPENT	12.1K	20.2K	0.3K	4K	12.4K	24.2K
TWOFISH	5.5K	13.6K	4.8K	14.5K	10.3K	28.1K



	Karol&Gorski's Program	Our Program	Karol&Gorski's Data	Our Data	Karol&Gorski's Total	Our Total
MARS	20	52	8	39	28	91
RC6	13	42	0	35	13	77
RIJNDAEL	25	53	33	46	58	99
SERPENT	43	72	1	14	44	86
TWOFISH	19	49	17	51	37	100

Figure 3.7. Relative Percentages of Memory Usage According to Implementation

When Figure 3.7 is investigated, following observations can be made for each algorithm:

MARS: Program memory usage for our implementation is 2.5 times of the previous one whereas data memory usage is 5 times of the previous one. The total memory usage is the three times of the previous one.

RC6: Program memory usage is 3 times of the previous one. Data memory usage is nearly 100 times of the previous one, which can be calculated from Table 5 by dividing 10027 to 104.

RIJNDAEL: Program memory usage is doubled whereas data memory usage is increased 40% for our implementation with respect to previous implementation.

SERPENT: Program memory usage is doubled. Data memory usage is 14 times of the previous implementation.

TWOFISH: In our implementation program memory usage is 2.5 times while data memory usage is the 3 times of the previous one.

When Table 3.11 is investigated, program memory usages of all algorithms for both implementations are less than 16 K * 16 bits, which is a moderate value for an implementation, except SERPENT, whose program memory value is 20699. If an external program memory is to be used for implementing the algorithms, it can be said that algorithms are suitable for implementation except SERPENT. SERPENT must be optimized for space to be implemented in this condition.

Data memory usages are below 16 K * 16 bits, which is a moderate value for an implementation. However, Table 3.11 shows that data memory usages are increased 100 times for RC6 and 13 times for SERPENT effecting the implementation of these algorithms in conditions where no external memory exists.

Table 3.13 shows that speed ranks of RIJNDAEL, SERPENT and TWOFISH remains the same, whereas MARS and RC6 change ranks and while the memory rank of RC6 remains the same, remaining algorithms' rank change completely.

Table 3.13. Comparison of Speed and Memory Ranks for Implementations

	SPEED		**MEMORY**	
ALGORITHM	**PREVIOUS IMP.**	**OUR IMP.**	**PREVIOUS IMP.**	**OUR IMP.**
MARS	4	3	2	3
RC6	3	4	1	1
RIJNDAEL	1	1	5	4
SERPENT	5	5	4	2
TWOFISH	2	2	3	5

Even though the used development environments are different, namely processor and the compiler version, indeed the processors in both environments have the same core, which is TMS320C54x. Only difference between TMS320C541 and TMS320C5416 is the peripheral difference, which is not affecting the cycle count or memory usage values. The main reason for having different values for cycle counts and memory usage is the compiler version used. The ways that the compilers generate executables from the given codes are different. The previous one, compiler version 1.20, generates executables in a way that minimizing the memory used but at the same time leading to high cycle count values whereas compiler with version 2.10 does the reverse.

To sum up, we can deduce that different development environments can lead different performance figures. And even for the same development environment, if different code generation tools are used the results obtained can be different. So when evaluating performance of these algorithms, the properties of the development environments and processors must be taken into account.

3.2. COMPARISON OF ALGORITHMS ACCORDING TO PLAINTEXT CHANGES

In this section, we investigate the dependency and the sensitivity of AES Finalist Algorithms to the plaintext changes. After explaining the objective of the analysis and the test method in section 3.2.1, we analyze each algorithm giving the maximum and minimum values of cycle counts for encryption and decryption operations in section 3.2.2.

3.2.1. Objective of Analysis and the Test Method

The AES Finalist Algorithms are investigated according to plaintext changes when the 128 bit long key is set to a fixed value. The objective for this analysis is to discover whether the encryption and decryption depend upon the plaintext itself and the how significant this dependency is.

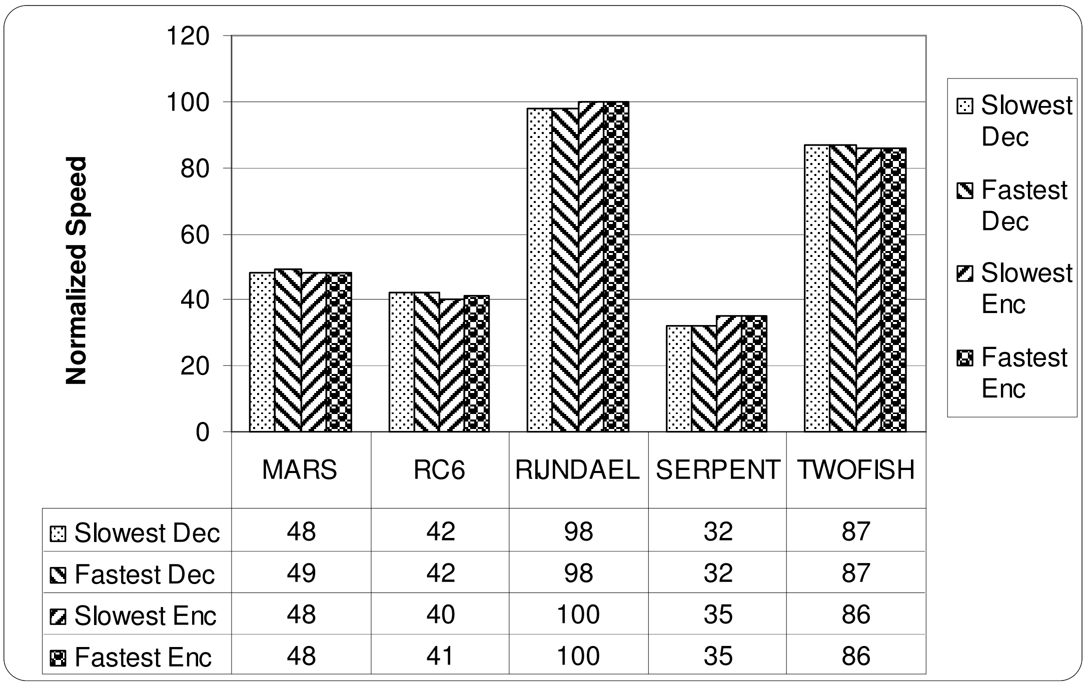
While making this analysis, the output of each encryption is used as the plaintext to the following encryption and after each encryption, a decryption is applied. The plaintext used as the starting plaintext and the key used throughout the whole test is a truly random 128 bit long sequence, which is “0x9F589F5CF6122C32B6BFEC2F2AE8C35A”.

3.2.2. Evaluation Results

In this section, after giving minimum and maximum cycle count values of encryption and decryption operations for all algorithms together with the relative percentage of speed comparison figure, which is obtained by normalizing these values according to minimum value, in Table 3.14 and Figure 3.8 respectively, we analyze each algorithm individually.

Table 3.14. Maximum and Minimum Cycle Count Values for Plaintext Changes

	DECRYPTION		**ENCRYPTION**	
	Max	***Min***	***Max***	***Min***
MARS	4116	4052	4144	4080
RC6	4735	4643	4936	4848
RIJNDAEL	2017	2017	1970	1970
SERPENT	6174	6174	5681	5681
TWOFISH	2252	2252	2296	2296



	MARS	RC6	RIJNDAEL	SERPENT	TWOFISH
▨ Slowest Dec	48	42	98	32	87
▨ Fastest Dec	49	42	98	32	87
▨ Slowest Enc	48	40	100	35	86
▨ Fastest Enc	48	41	100	35	86

Figure 3.8 Relative Percentages of Speed Comparison According to Plaintext Changes

After examining Table 3.14 and Figure 3.8, we reach to the following results for each algorithm:

MARS:

As seen from the Table 3.14, the speeds of both encryption and decryption depend on the plaintext used, namely they change with the changing plaintext. Figure 3.8 shows that the changes in the speeds are only 1% and less than 1% for encryption and decryption operations respectively.

The reason for variable speeds for encryption and decryption is data dependent rotations used in “E Function “of encryption and decryption operations, [Burwick, 1999].

RC6:

It is evident from Table 3.14 that the encryption and decryption operations depend on plaintext. They change with the changing plaintext. Dependence of the encryption and decryption speeds to the plaintext are due to the data dependent rotations used in both operations [Rivest, 1998]. When Figure 3.8 is investigated, it is seen that amount of speed variation depending upon the plaintext change is less than 1%.

RIJNDAEL:

If minimum and maximum values are investigated in Table 3.14, it is obvious that encryption and decryption speeds do not change, as the slowest and fastest speed values are the same for both operations. Because RIJNDAEL has no data dependent operations, both encryption and decryption are independent of plaintext changes [Daemen, Rijmen, 1999].

SERPENT:

The minimum and the maximum values of cycle counts are the same, see Table 3.14. This shows neither encryption nor decryption depends upon plaintext. Since SERPENT does not have any data dependent operations in either encryption or decryption, cycle count values for these operations do not change with the changing plaintext.

TWOFISH:

Like RIJNDAEL and SERPENT, the minimum and the maximum values of cycle counts for encryption and decryption operations are the same, see Table 3.14. This shows neither encryption nor decryption depends upon plaintext. Since TWOFISH does not have any data dependent operations in both encryption and decryption, these operations are insensitive to plaintext changes.

In conclusion, we can say that encryption and decryption speeds show a variable characteristic for MARS and RC6 when the plaintext is changed. This is due to the data dependent operations in these algorithms. Since RIJNDAEL, SERPENT, and TWOFISH do not have any data dependent operations in either encryption or decryption, cycle count values for these operations do not change with the changing plaintexts. Therefore, we can say that these algorithms are insensitive to plaintext changes.

3.3 ANALYSIS OF ALGORITHMS ACCORDING TO KEY CHANGES

In this section, we investigate the dependency and the sensitivity of AES Finalist Algorithms to the key changes. After explaining the test method and objective of the analysis in section 3.3.1, we analyze each algorithm giving the maximum and minimum values of cycle counts for encryption, decryption and key setup operations in section 3.3.2.

3.3.1 Objective of the Analysis and the Test Method

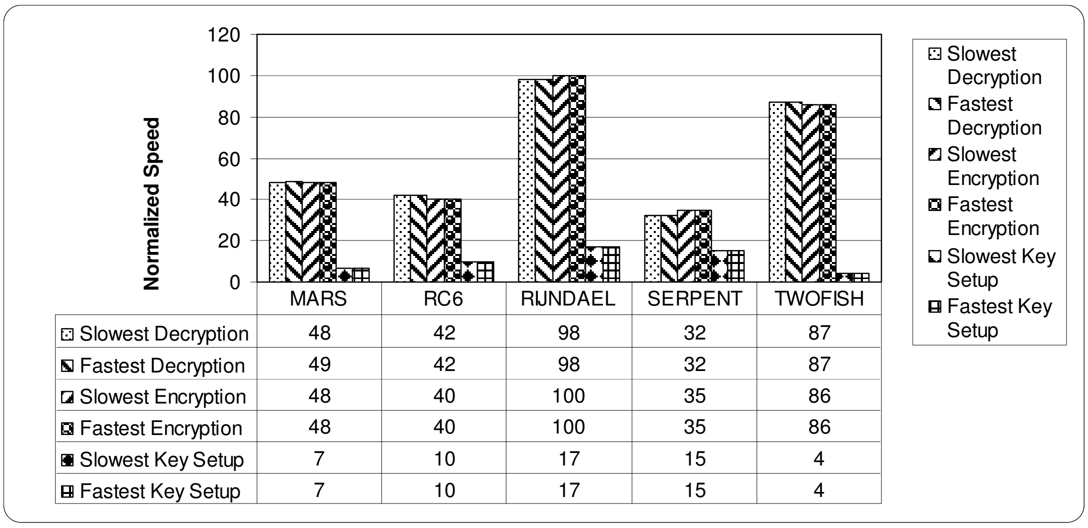
While analyzing the sensitivity of algorithms to key changes, the plaintext is set to a fixed value and key is changed in every iteration. The key used in each iteration is the output of the previous encryption. After every encryption, a decryption is applied. The objective for this analysis is to find out whether the encryption, decryption and key setup show variable characteristics, when variable keys are used.

3.3.2 Evaluation Results

In this section, maximum and minimum cycle count values of encryption, decryption and key setup operations for each algorithm which indicate the best and worst cases respectively for each operation and relative percentage figure which is obtained by normalizing these maximum and minimum values according to minimum value are given.

Table 3.15. Maximum and Minimum Cycle Count Values for Key Changes

	DECRYPTION		ENCRYPTION		KEY SETUP	
	Max	***Min***	***Max***	***Min***	***Max***	***Min***
MARS	4104	4056	4132	4084	28427	28169
RC6	4723	4663	4920	4868	20661	20549
RIJNDAEL	2017	2017	1970	1970	11464	11464
SERPENT	6174	6174	5681	5681	13512	13512
TWOFISH	2252	2252	2296	2296	51733	51688



	MARS	RC6	RIJNDAEL	SERPENT	TWOFISH
Slowest Decryption	48	42	98	32	87
Fastest Decryption	49	42	98	32	87
Slowest Encryption	48	40	100	35	86
Fastest Encryption	48	40	100	35	86
Slowest Key Setup	7	10	17	15	4
Fastest Key Setup	7	10	17	15	4

Figure 3.9. Relative Percentages of Speed Comparison According to Key Changes

We can deduce the following results and conclusions from Table 3.15 and Figure 3.9 for each algorithm:

MARS:

Table 3.15 shows that; encryption, decryption and key setup cycle count values change with changing key. Although the operations depend on key, percentage of change is only 1%, see Figure 3.9. Since encryption, decryption and key setup have data dependent operations in which the key is used in a way that affecting execution speed, we obtain variable cycle count values for these operations with the changing key.

<u>RC6:</u>

Like MARS, RC6's encryption, decryption and key setup are also dependent on the key used, see Table 3.15. Figure 3.9 shows that the percentage variations of encryption, decryption and key setup speeds are less than 1%. Although a fixed plaintext is used throughout the whole test, since encryption and decryption have data dependent operations, where the key takes place, [Rivest, 1998], these operations have variable cycle count values.

<u>RIJNDAEL:</u>

Table 3.15 shows that encryption, decryption, and key setup operations of RIJNDAEL are insensitive to key changes. When RIJNDAEL is investigated for a fixed plaintext, it is obvious that the cycle count values for all operations do not change with changing key. This is due to key independent structure of RIJNDAEL [Daemen, Rijmen, 1999].

<u>SERPENT:</u>

SERPENT shows the same characteristics as RIJNDAEL. None of the operations depends on the key used. It gives the same cycle count values for all keys, see Table 3.15. Like RIJNDAEL, SERPENT also has a key independent structure [Anderson, Biham, Knudsen, 1998].

<u>TWOFISH:</u>

Encryption and decryption operations of TWOFISH do not depend on the key used, but the key setup operation depends (see Table 3.15). The variation of cycle count value for key setup operation is related with the constitution of key dependent S-boxes [Schneier, 1998].

To sum up, both MARS and RC6 have variable key setup, encryption, and decryption operations due to data dependent rotations related to key. RIJNDAEL and SERPENT do not have any data dependent operations so their cycle counts remain constant with the changing key. TWOFISH's encryption and decryption cycle count value are independent of key, but since variable S-boxes are constructed in key setup operation, its key setup has a variable cycle count value. The amounts of dependencies of MARS, RC6, and TWOFISH to key changes are in the level of 1%.

3.4 ANALYSIS OF ALGORITHMS ACCORDING TO KEY LENGTH CHANGES

The sensitivity of AES finalist algorithms to key length changes are investigated in this part. After giving the test method and the objective of this analysis in section 3.4.1, we analyze each algorithm separately in section 3.4.2

3.4.1 Objective of the Analysis and the Test Method

Objective for analyzing the algorithms for the key length changes is to find out the characteristics of algorithms when varying keys are used.

To discover the sensitivity of algorithms to key length changes, all of the algorithms are run for 128-bit, 192-bit, and 256-bit long keys. For each key length 100 encryption, decryption, and key setup operations are done and the average cycle count values are tabulated in the following section.

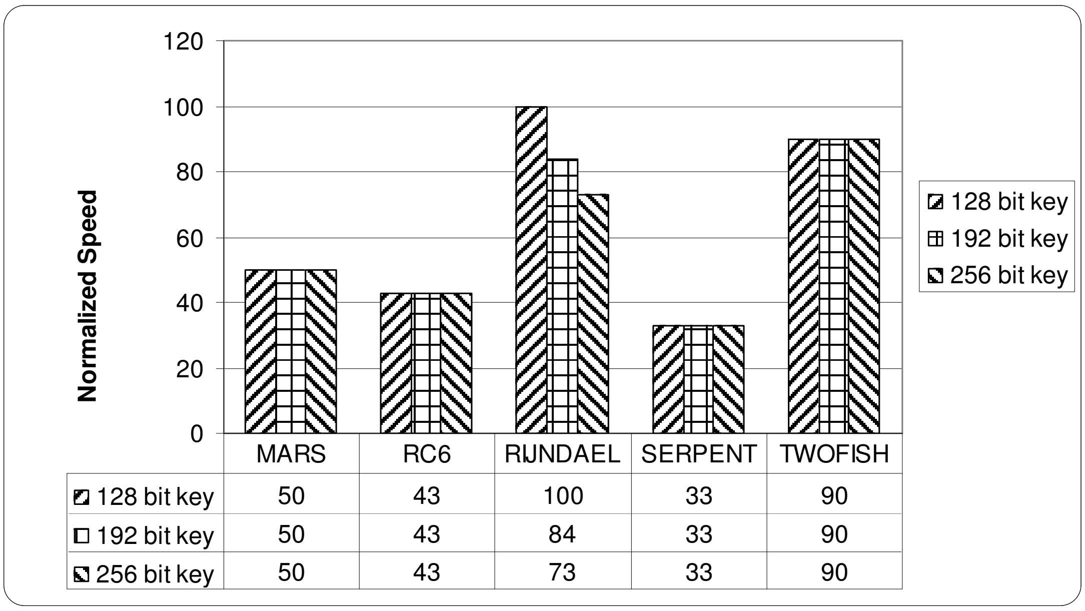
At the beginning, the plaintext is set to a fixed value which is "0x11223344112233441122334411223344" for all key lengths and the key is set to
"0x11223344112233441122334411223344",
"0x112233441122334411223344112233441122334411223344",
"0x1122334411223344112233441122334411223344112233441122334411223344", for 128-bit, 192-bit, and 256-bit respectively at the beginning. After each encryption, the cipher text, which is 128-bit long, is used as the key for the next iteration. For 192-bit and 256-bit keys the least significant 64 bits and 128 bits are left as they are (0x1122334411223344 for 192-bit and 0x11223344112233441122334411223344 for 256-bit).

3.4.2 Evaluation Results

In this section, average cycle count values of encryption, decryption and key setup operations for each algorithm together with the relative percentage figures, which are obtained by normalizing these values according to minimum values of decryption, encryption and key setup, are given.

Table 3.16. Variation of Decryption Cycle Count According to Key Length Change

	128 bit key	*192 bit key*	*256 bit key*
MARS	4056	4056	4056
RC6	4663	4663	4663
RIJNDAEL	2017	2394	2770
SERPENT	6174	6174	6174
TWOFISH	2252	2252	2252



	MARS	RC6	RIJNDAEL	SERPENT	TWOFISH
▨ 128 bit key	50	43	100	33	90
□ 192 bit key	50	43	84	33	90
▧ 256 bit key	50	43	73	33	90

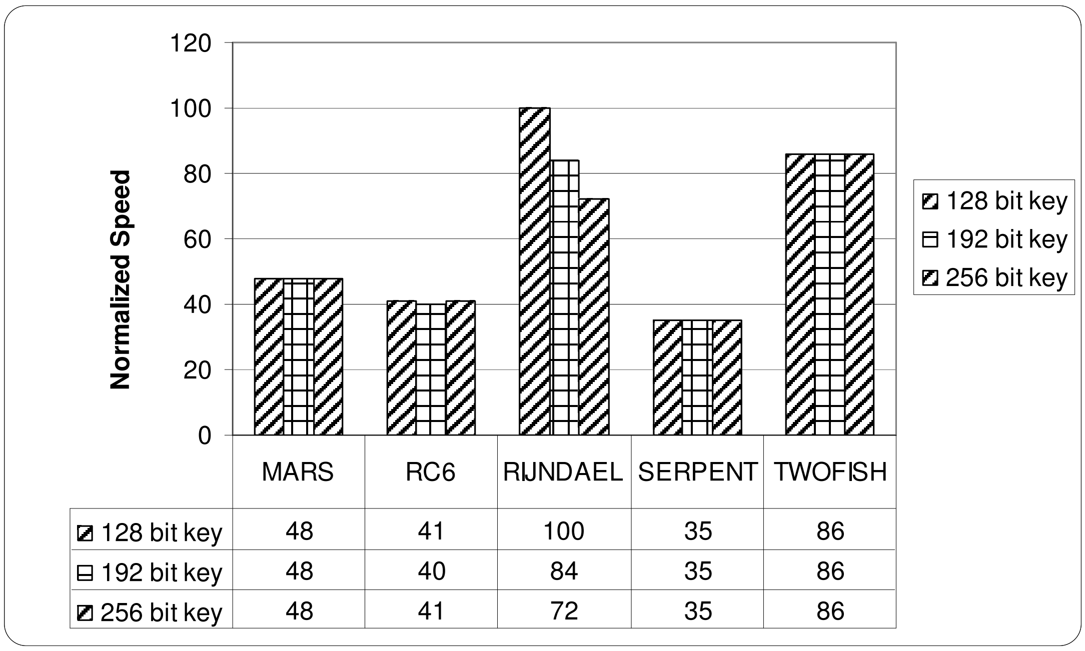
Figure 3.10. Relative Percentages of Speed Variation of Decryption According to Key Length Change

Table 3.16 and Figure 3.10 show that decryption speeds of MARS, RC6, SERPENT and TWOFISH do not change with the changing key length whereas RIJNDAEL shows a significant amount of change with the changing key length.

Although MARS and RC6 has data dependent rotations in decryption, their decryption are dependent on the key rather than its length. So changing key length does not change the cycle count values. Since SERPENT and TWOFISH do not have any data dependent operation in their decryption operation, changing key length does not change the cycle count for decryption. The number of rounds increases when the key length increases for RIJNDAEL, which is 10, 12 and 14 for 128-bit, 192-bit and 256-bit long key respectively. This leads variable cycle count values for decryption of RIJNDAEL.

Table 3.17. Variation of Encryption Cycle Count According to Key Length Change

	128 bit key	***192 bit key***	***256 bit key***
MARS	4084	4084	4084
RC6	4856	4868	4856
RIJNDAEL	1970	2347	2723
SERPENT	5681	5681	5681
TWOFISH	2296	2296	2296



	MARS	RC6	RIJNDAEL	SERPENT	TWOFISH
▨ 128 bit key	48	41	100	35	86
⊟ 192 bit key	48	40	84	35	86
▨ 256 bit key	48	41	72	35	86

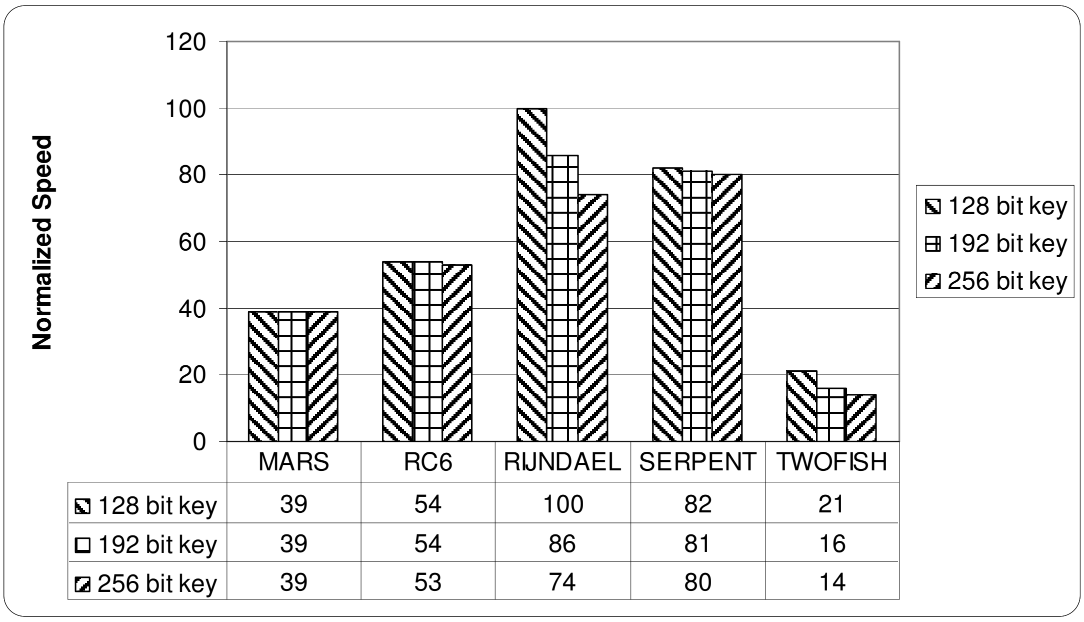
Figure 3.11. Relative Percentages of Speed Variation of Encryption According to Key Length Change

Table 3.17 and Figure 3.11 show that decryption speeds of MARS, SERPENT and TWOFISH do not change with the changing key length whereas RIJNDAEL shows a significant amount of change with the changing key length. In addition, RC6's encryption speed changes slightly with the changing key length.

As decryption, MARS's and RC6's encryptions have data dependent rotations, their encryption are dependent on the key rather than its length. So changing the key length does not change their average cycle count values. The slight decrease of cycle count for 192-bit encryption of RC6 is because of the difference of the keys used in the experiment. Since SERPENT and TWOFISH do not have any data dependent operation in their encryption operation, changing key length does not change the cycle count for encryption. The number of rounds increases when the key length increases for RIJNDAEL, which is 10, 12 and 14 for 128-bit, 192-bit and 256-bit long key respectively. This leads variable cycle count values for encryption of RIJNDAEL as decryption.

Table 3.18. Variation of Key Setup Cycle Count According to Key Length Change

	128 bit key	*192 bit key*	*256 bit key*
MARS	28169	28172	28174
RC6	20549	20612	20678
RIJNDAEL	11044	12888	14984
SERPENT	13512	13658	13768
TWOFISH	51749	67942	80755



	MARS	RC6	RIJNDAEL	SERPENT	TWOFISH
▧ 128 bit key	39	54	100	82	21
□ 192 bit key	39	54	86	81	16
▨ 256 bit key	39	53	74	80	14

Figure 3.12. Relative Percentages of Speed Variation of Key Setup According to Key Length Change

Table 3.18 shows that MARS and RC6 have almost constant key setup cycle count values for different key lengths. Figure 3.12 shows that the significance of variation in key setup speeds are less than 1% for MARS and are equal to 1% for RC6. The reason for slight variation of key setup cycle count of MARS is due to differences of the keys rather than the key length. But it is due to increase of operations in RC6. RIJNDAEL's key setup is highly dependent on the key length because of the increase of the required subkeys with the increasing rounds. SERPENT shows a variable characteristic for changing key lengths due to increasing processing load with the increasing key length. SERPENT's key setup speed decreases in an amount of 1% and 2% for 192-bit and 256-bit key as compared to 128-bit key.TWOFISH key setup speed value also changes with the changing key length significantly like RIJNDAEL because of the increase of key dependent s-box number with increasing key length. Number of S-box is 2, 3, and 4 for 128-bit, 192-bit and 256-bit respectively.

Encryption, decryption, and key setup operations of MARS and RC6 do not depend on the length of the key used. SERPENT encryption and decryption do not depend on the key length but its key setup speed decreases as the key length increases because of the key initialization. RIJNDAEL has variable cycle counts for all operations due to increasing of processes and is strongly dependent on the key length. TWOFISH's encryption and decryption also do not depend on the key length, but because of S-box construction in the key setup, this operation is strictly dependent on the key length.

CHAPTER 4

IMPLEMENTATION OF MARS, RC6 AND RIJNDAEL CIPHERS IN TMS320C5416 ASSEMBLER LANGUAGE

In this chapter, implementation of MARS, RC6 and RIJNDAEL ciphers in TMS320C5416 assembler language is explained. Cycle counts achieved in assembler language are compared with those of the C codes written by Brian Gladman [Gladman, 1999].

4.1 IMPLEMENTING MARS, RC6 AND RIJNDAEL CIPHERS IN TMS320C5416 ASSEMBLER LANGUAGE

We start the implementation of MARS, RC6 and RIJNDAEL ciphers in TMS320C5416 assembler language by first investigating the algorithm structures. All of the algorithms are divided into three parts, which are encryption, decryption and key setup. Each part is defined as a C function and the operations in each part defined as an assembler function excluding the variable initializations and table generations since it is more flexible and easy to define and initialize variables in C language. Another advantage of defining assembler codes in C code is that it is possible to use Stack Pointer Addressing mode of DSP, [TMS320C54x Ref, 1996], which provides to write optimum codes.

General code structure can be explained as follows: We start writing code by first defining the tables, which can be S boxes or initializing tables, and then we define variables used in the C code. Table generating function takes place after these definitions, which is used only for Rijndael. After definitions, the main function calling the key setup, encrypt, and decrypt functions is placed. In key setup function, after making variable initializations, assembler code realizing the key setup process is defined and the return values are processed to generate round keys. The general pseudo code for the code structure is as follows:

```
Table definitions //C code
Variable definitions //C code
Table generating functions(used for only RIJNDAEL) //C code
void main (void) //C code
{
        set the input key //C code
        set the input plaintext //C code
        keysetup(key); //C code
        encrypt(plaintext,key); //C code
        decrypt(ciphertext,key); //C code
}
keysetup(key) //C code
{
        variable initializations //C code
        keysetupasm(input); //Assembler code called from C code
        variable  finalizations //C code
}
```

```
encrypt(plaintext,key) //C code
{
        variable initializations //C code
        encryptasm(input); //Assembler code called from C code
        variable  finalizations //C code
}
decrypt(ciphertext,key) //C code
{
        variable initializations //C code
        decryptasm(input); //Assembler code called from C code
        variable  finalizations //C code
}
```

After implementing the ciphers for 128-bit key length, we modify the algorithms for 192 and 256-bit long keys. In these modifications, while MARS and RC6 do not require many changes, only in key setup, RIJNDAEL requires more changes as compared to MARS and RC6 due to increasing rounds in all operations.

4.1.1 Implementing MARS

We start implementing MARS by first defining the s-box used in all operations and the initial values used in key setup in the C code. After defining the input key and the initial values as four 32-bit words in the main function, we pass them to the C code of encryption, decryption and key setup. In encryption and decryption after doing the initial key additions to these four variables, we pass the addresses of these variables to the assembler code together with the address of the round key. After processing the inputs in encryption and decryption, we make the final additions in the C code. In key setup, after processing the inputs, we do not apply any operation to the return variables.

We first implement the assembler code of encryption since it is almost the same as decryption. While implementing the encryption, we use direct implementation without using any loops. This provides an optimum code. We implement every distinct operation once and repeat every operation as it is required. We do not use any loop to repeat these operations. Decryption is very similar to encryption and was implemented using a similar structure as encryption. Different from encryption and decryption we use loops while implementing key setup operation. (See Appendix A.1 for encryption and decryption flowcharts. See Appendix D, in CD, for codes of MARS)

4.1.2 Implementing RC6

We start implementing RC6 by first defining the round key globally in the C code. In the main function, we define input key and plaintext as four 32-bit words and then pass them to encryption, decryption and key setup functions. In encryption and decryption, after making the key additions to input plaintext, we pass the pointers of these words into the assembler function. And after processing the words in assembler, we make the final key additions in the C code. For key setup, after processing the four 32-bit input key in C code, we give the pointers of processed inputs to the assembler code. But we do not apply any operation to the return values of the assembler code.

Like MARS, encryption and decryption operations are nearly the same. So we start with the encryption and then write decryption by making slight modifications on encryption. While implementing the key setup, we use loops rather than explicitly writing each iteration of round function. (See Appendix A.2 for encryption and decryption flowcharts. See Appendix D, in CD, for codes of RC6)

The key point in implementing the encryption in assembler in an optimized way is to reduce the time consumed by multiplication operation, which is widely used in both encryption and decryption, [Rivest, 1998]. Therefore we utilize the “multiply and accumulate” utility of the DSP, [TMS320C54X Ref, 1996], which can be completed one cycle. Thus we are able reduce the time consumed by multiplications significantly. The code for realizing multiplication in an efficient way is in Appendix B.1.

4.1.3 Implementing RIJNDAEL

We begin implementing RIJNDAEL by defining global variables and table generating functions. In the main function, we define input key and plaintext as four 32-bit words and pass them to encryption decryption and key setup functions. After making the initial XOR operations in C code, we pass the addresses of these four variables to the assembler codes of encryption, decryption and key setup functions. As MARS and RC6, RIJNDAEL's encryption and decryption operations are almost the same. Therefore, we implement the encryption first and then adapt decryption accordingly.

Encryption is implemented using the efficient implementation procedure (See Appendix C) as suggested by the algorithm's designers, [Daemen, Rijmen, 1999]. In this implementation, rather than using key addition, s-box look-up, column mixing and row shifting in a straight forward manner, we reduce the implementation to simple table look-ups, which are produced by the C code, and XOR operations.

While implementing encryption and decryption, the rounds are implemented using loops but then they are coded explicitly by writing each round one by one (See Appendix A.3 for encryption and decryption flowcharts). While writing each round function, we used the circular addressing, [TMS320C54X Ref, 1996], mode of DSP to increase the efficiency (See Appendix B.2 for table lookup code. See Appendix D, in CD, for codes of RIJNDAEL).

We implement the key setup operation by using loop concept. The multiplication operations in the key setup are coded by utilizing "multiply and accumulate" property of DSP to increase efficiency.

4.2 ASSEMBLER IMPLEMENTATION RESULTS

After implementing the algorithms in assembler language, we measure the performances by using 100 random key and plaintext combinations for all key lengths. The comparison of the results of assembler codes with the C codes written by Brian Gladman, [Gladman, 1999], for encryption, decryption and key setup are given in Table 4.1, 4.2 and 4.3 respectively.

Table 4.1. Number of Machine Cycles Required for Encryption

Key Length		**Encrypt in C**	**Encrypt in Assembler**	**Improvement Ratio**
128 bit	**MARS**	4110	3600	13%
	RC6	4889	3554	28%
	RIJNDAEL	1970	1579	20%
192 bit	**MARS**	4084	3598	12%
	RC6	4868	3523	28%
	RIJNDAEL	2347	1864	21%
256 bit	**MARS**	4084	3600	12%
	RC6	4856	3540	28%
	RIJNDAEL	2723	2149	22%

Table 4.1 shows that the encryption implemented in assembler language has better performance with respect to C implementation. Ratio of improvement is the maximum for RC6, while it is the minimum for MARS. Improvement amounts of algorithms remain nearly the same for all key lengths. RC6's improvement ratio is around 30% while RIJNDAEL's ratio is around 20%. MARS's improvement ratio is around 10%.

Table 4.2. Number of Machine Cycles Required for Decryption

Key Length		Decrypt in C	Decrypt in Assembler	Improvement Ratio
128 bit	**MARS**	4082	3584	13%
	RC6	4694	3748	21%
	RIJNDAEL	2017	1698	16%
192 bit	**MARS**	4056	3582	12%
	RC6	4663	3717	21%
	RIJNDAEL	2394	1999	17%
256 bit	**MARS**	4056	3584	12%
	RC6	4663	3735	20%
	RIJNDAEL	2773	2300	18%

Table 4.2 shows that the decryption implemented in assembler language has better performance with respect to C implementation like encryption. Ratio of improvement is the maximum for RC6, while it is the minimum for MARS. Improvement amounts of algorithms remain nearly the same for all key lengths. RC6's improvement ratio is around 20% while RIJNDAEL's ratio is around 17%. MARS's improvement ratio is around 10%. Improvement ratio of RIJNDAEL is much closer to RC6 in this case.

The amounts of improvement between encryption and decryption operations of RC6 and RIJNDAEL are different. The improvement amount for decryption is less than encryption for both algorithms. The reason for this difference is the extra branch operations used in decryption of RC6 and is the extra operations used for key arrangement in each round of encryption. Another reason can be that we write decryption codes after writing encryption codes and try to adapt. If we started writing encryption first and then decryption, then the situation could be reverse.

Table 4.3. Number of Machine Cycles Required for Key Setup

Key Length		Key Setup in C	Key Setup in Assembler	Improvement Ratio
128 bit	MARS	28220	8039	72%
	RC6	20595	19346	7%
	RIJNDAEL	11464	5252	64%
192 bit	MARS	28174	7691	72%
	RC6	20612	19347	6%
	RIJNDAEL	12888	6499	50%
256 bit	MARS	28174	7893	72%
	RC6	20678	19389	6%
	RIJNDAEL	14984	7479	50%

Table 4.3 shows that assembler implementations of key setup have better performances as compared to C implementation. The amount of improvement is very high for MARS and RIJNDAEL. Improvement ratio is around 70% for MARS and around 50-60% for RIJNDAEL while it is only around 5% for RC6. The improvement ratios remain nearly the same for all key lengths.

In a typical session, key setup is made for once at the beginning of the session and the round keys are used throughout the operation. Therefore, key setup speed does not affect the speed of session as much as encryption and decryption. Hence, we can say that we are able to increase the speed of session around 13%, 21-28%, and 16-20% for MARS, RC6 and RIJNDAEL respectively.

CHAPTER 5

CONCLUSIONS

In this thesis, implementation properties of AES Finalist Algorithms are examined in the sense of development environment, sensitivities to plaintext, key and key length variations by relating the reasons of these variations to the structural properties of algorithms. The C codes written by Brian Gladman [Gladman, 1999] are adapted to TMS320C54X, and then the speed and memory usage values are compared with the adaptation of Karol Gorski and Michal Skalski's [Gorski, Skalski, 1999]. Three of the algorithms, MARS, RC6 and RIJNDAEL, are implemented on TMS320C54X by using the assembler language. The results show that assembler implementations are improved with respect to C implementations by 13% for MARS, 16-20% for RIJNDAEL and 21-28% for RC6.

The properties of development environment affect the evaluation results dramatically. Namely, the structure of the processor, i.e. the processors' processing units, which can be 8/16/32/64 bits, or its MIPS (Million Instructions Per Seconds) value or its ability to do specific operations like 32 bit multiplication and the tools to generate executables from written codes play very important roles in the ranking of the algorithms in both cycle count and memory usage respects. However, the development environment has no effect on the results of structural behaviors of algorithms like the algorithms' dependency on the key, plaintext or key length.

In this study, TMS320C5416 DSK Stand-Alone Development and Evaluation Module is used as the development environment. Speed and memory usage values are measured for algorithms. For all algorithms, speeds of almost all operations for our implementation are doubled with respect to Karol Gorski and Michal Skalski's implementation on the same processor with a different compiler, which uses the same C codes written by Brian Gladman as in our adaptation. However, the total memory usages are increased in the range of 2.5 and 3 times. Speed ranks of RIJNDAEL, SERPENT and TWOFISH remain the same, whereas MARS and RC6 change ranks. While the memory rank of RC6 remains the same, those of the remaining algorithms change completely.

When the algorithms are investigated according to their dependencies on plaintexts, keys, and key lengths, the results can be summarized for each algorithm as follows:

MARS' encryption and decryption operations are both key and plaintext dependent operations. Namely, when either plaintext or key is changed, the amount of spent cycle counts, and hence the time, changes. This is due to the "E function" residing in both operations. Its key setup operation is also dependent on the key. For different key lengths, the key setup, encryption, and decryption operations depend on the key length because of the varying key with the varying key length. However, the average values of cycle counts obtained after 100 iterations for encryption, decryption and key setup operations remain the same for varying key lengths. Although MARS' encryption, decryption, and key setup are plaintext and key dependent operations, the amount of speed differences are in level of 1%.

Encryption and decryption speeds show a variable characteristic for RC6, when the key length is set to a fixed value and either the key or the plaintexts are changed. This is due to the data dependent operations in these operations. The key setup operation is also key dependent. The variations in the cycle counts in encryption, decryption and key setup operations are in level of 1%. Encryption, decryption and key setup are key dependent operations, so changing key length also changes the cycle count values, but the average value of cycle counts obtained after 100 iterations of these operations remain the same for varying key lengths.

RIJNDAEL’s encryption, decryption, and key setup operations are independent from key and plaintext. However, when the key length is increased, the speeds of these operations decrease in the range of 15% and 28% for 192-bit and 256-bit keys respectively as compared to 128-bit key. This arises from the increase of the number of rounds, which is 10, 12 and 14 for 128-bit, 192-bit and 256-bit keys respectively, with increasing key length.

SERPENT’s encryption and decryption are not affected from key, plaintext, and key length changes because of its key and plaintext independent structure. However, SERPENT’s key setup speed decreases due to initialization of key at the beginning of key setup in an amount of 1% and 2% for 192-bit and 256-bit key as compared to 128-bit key.

TWOFISH’s encryption and decryption are not affected from key, plaintext, and key length changes, but its key setup is dependent on both key and key length. The amount of variation in the key setup speed due to key variation is in the range of 1%. Key setup speed varies when key length is changed in an amount of 13% and 23% for 192-bit and 256-bit key as compared to 128-bit key due to construction of S-boxes. Key setup generates 2, 3 and 4 S-boxes for 128-bit, 192-bit and 256-bit key respectively.

After comparing algorithms in C language, we implement MARS, RC6, and RIJNDAEL in TMS320C54X assembler language and compare the speed performances with those of the C implementation. The assembler implementations speed values are faster than the C implementation. MARS’ speed performance is increased around 13%, RIJNDAEL’s speed performance is increased around 16-20% and RC6’s speed performance is increased around 20-28%. Speed improvement can be further increased by using different approaches like eliminating inefficient operations and using more efficient implementation methods.

REFERENCES

[Anderson, Biham, Knudsen, 1998] R. Anderson, E. Biham, and L. Knudsen, *Serpent: A Proposal for the Advanced Encryption Standard*, AES algorithm submission, June 1998.

[Biham, Shamir, 1993] Eli Biham, Adi Shamir, Differential Cryptanalysis of the Data Encryption Standard, Springer Verlag, 1993. ISBN 0-387-97930-1, ISBN 3-540-97930-1.

[Burwick, 1999] C. Burwick, et al., *MARS – A Candidate Cipher for AES*, AES algorithm submission, August 20, 1999.

[Daemen, Rijmen, 1999] J. Daemen and V. Rijmen, *AES Proposal: Rijndael*, AES algorithm submission, September 3, 1999

[Feistel, 1973] H. Feistel. Cryptography and Computer Privacy. Scientific American, 228(5):15-23, May 1973.

[Gladman,1999] B.Gladman http://archive.devx.com/sourcebank/, January, 1999

[Gorski, Skalski, 1999] K. Gorski and M. Skalski, *Comments on AES Candidates*, AES Round 1 public comment, April 15, 1999

[Matsui, 1993] Linear cryptanalysis method for DES cipher. EUROCRYPT 1993

[Rivest, 1998] R. Rivest, et al., *The RC6™ Block Cipher*, AES algorithm submission, June 1998.

[Schneier, 1998] B. Schneier, et al., *Twofish: A 128-Bit Block Cipher*, AES algorithm submission, June 15, 1998

[Tms320c54x Ref, 1996] Texas Instruments, TMS320C54X DSP Reference Set Volume 1 CPU and Peripherals, October 1996

APPENDIX A

FLOWCHARTS OF ASSEMBLER IMPLEMENTATIONS OF ENCRYPTION AND DECRYPTION FUNCTIONS

A.1 FLOWCHARTS OF MARS

A.1.1 Flowchart Of Encryption

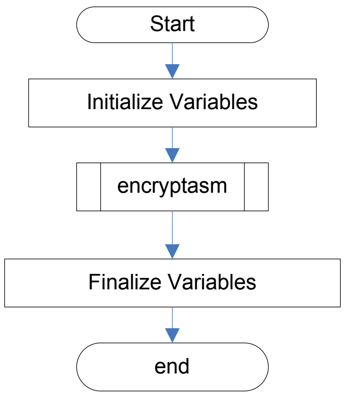


Figure A.2. Flowchart of encrypt

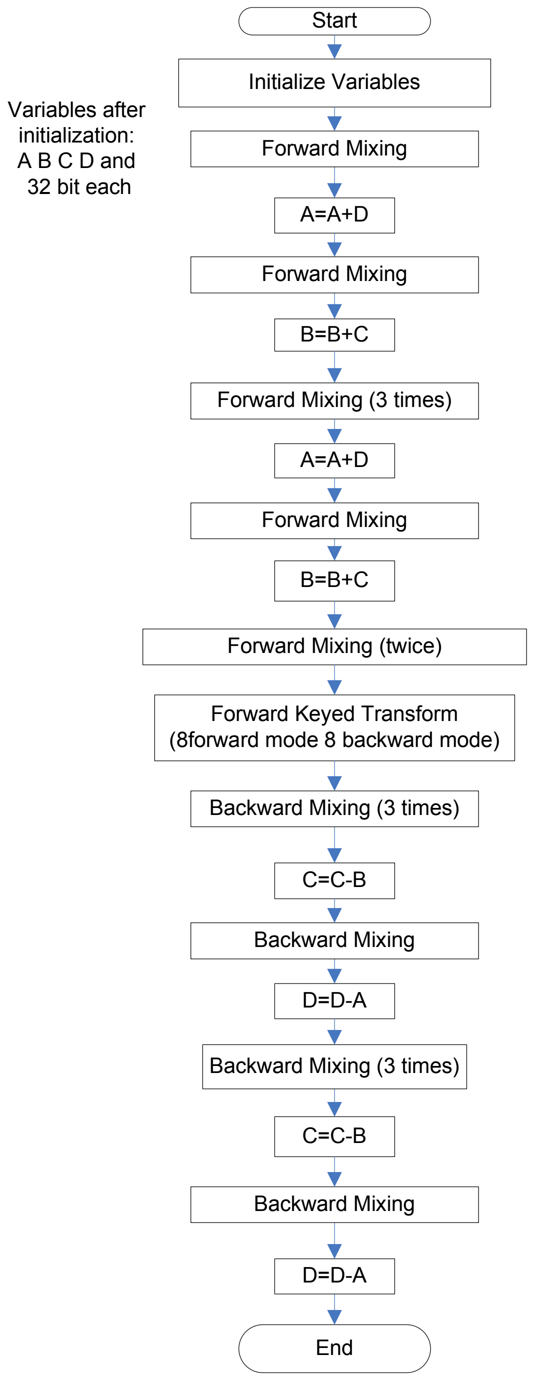


Figure A.2. Flowchart of encryptasm of MARS

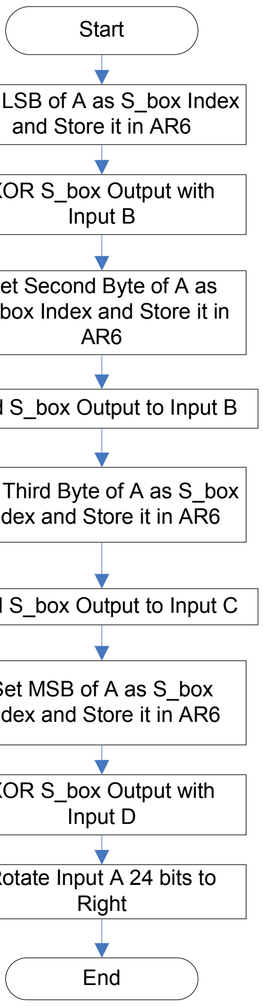

Start
Set LSB of A as S_box Index and Store it in AR6
XOR S_box Output with Input B
Set Second Byte of A as S_box Index and Store it in AR6
Add S_box Output to Input B
Set Third Byte of A as S_box Index and Store it in AR6
Add S_box Output to Input C
Set MSB of A as S_box Index and Store it in AR6
XOR S_box Output with Input D
Rotate Input A 24 bits to Right
End

Figure A.3. Flowchart of Forward Mixing

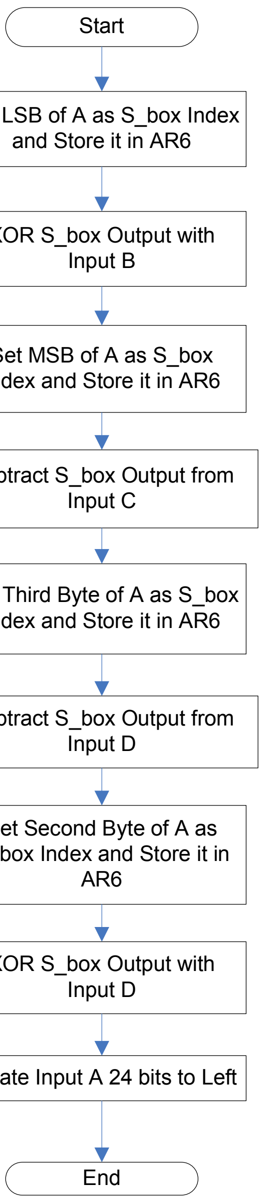

Start
Set LSB of A as S_box Index and Store it in AR6
XOR S_box Output with Input B
Set MSB of A as S_box Index and Store it in AR6
Subtract S_box Output from Input C
Set Third Byte of A as S_box Index and Store it in AR6
Subtract S_box Output from Input D
Set Second Byte of A as S_box Index and Store it in AR6
XOR S_box Output with Input D
Rotate Input A 24 bits to Left
End

Figure A.3. Flowchart of Backward Mixing

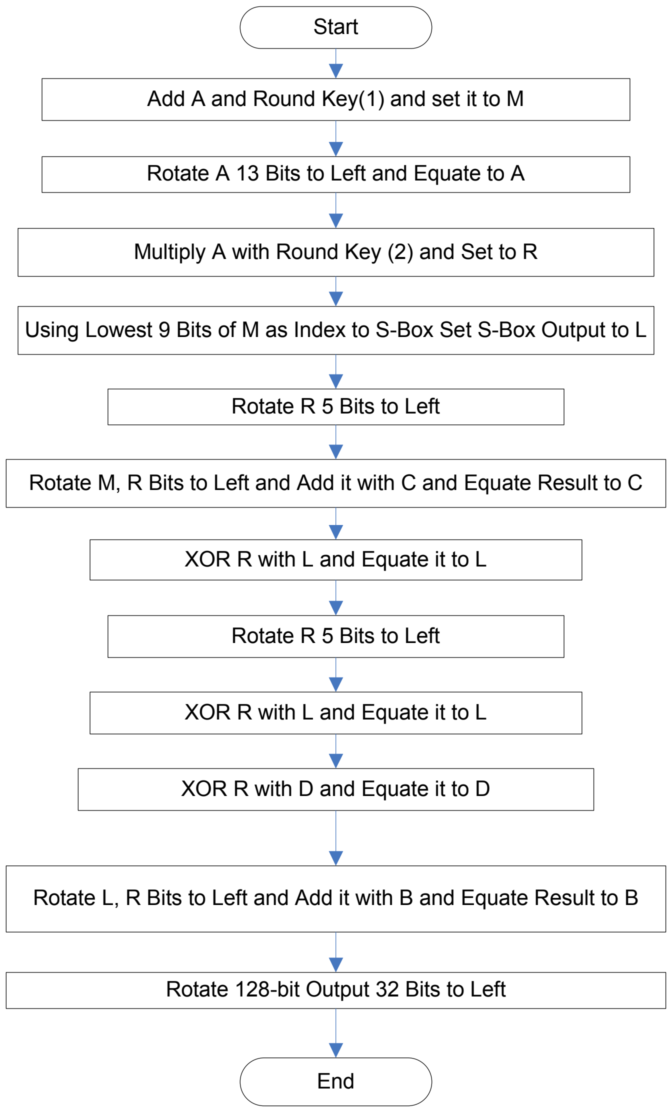

Start
Add A and Round Key(1) and set it to M
Rotate A 13 Bits to Left and Equate to A
Multiply A with Round Key (2) and Set to R
Using Lowest 9 Bits of M as Index to S-Box Set S-Box Output to L
Rotate R 5 Bits to Left
Rotate M, R Bits to Left and Add it with C and Equate Result to C
XOR R with L and Equate it to L
Rotate R 5 Bits to Left
XOR R with L and Equate it to L
XOR R with D and Equate it to D
Rotate L, R Bits to Left and Add it with B and Equate Result to B
Rotate 128-bit Output 32 Bits to Left
End

Figure A.4. Flowchart of Forward Keyed Transform

A.1.2. Flowchart of Decryption

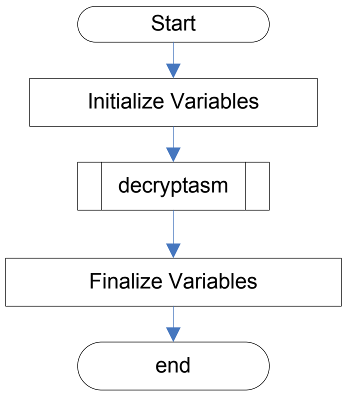


Figure A.5. Flowchart of Decryption

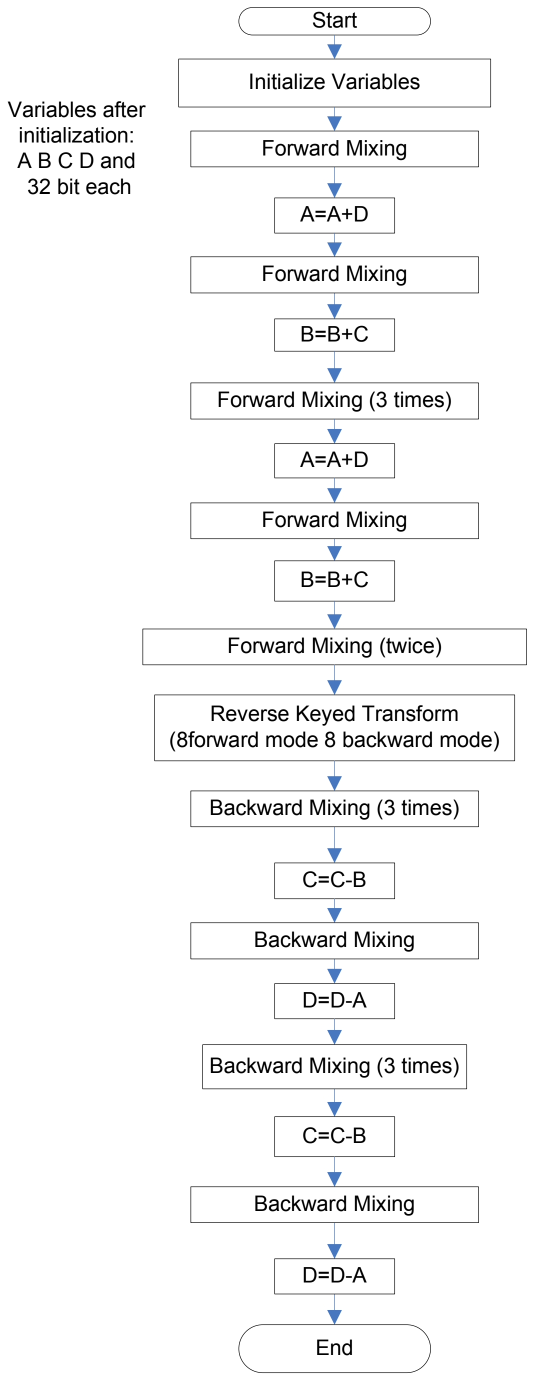


Figure A.6. Flowchart of decryptasm of MARS

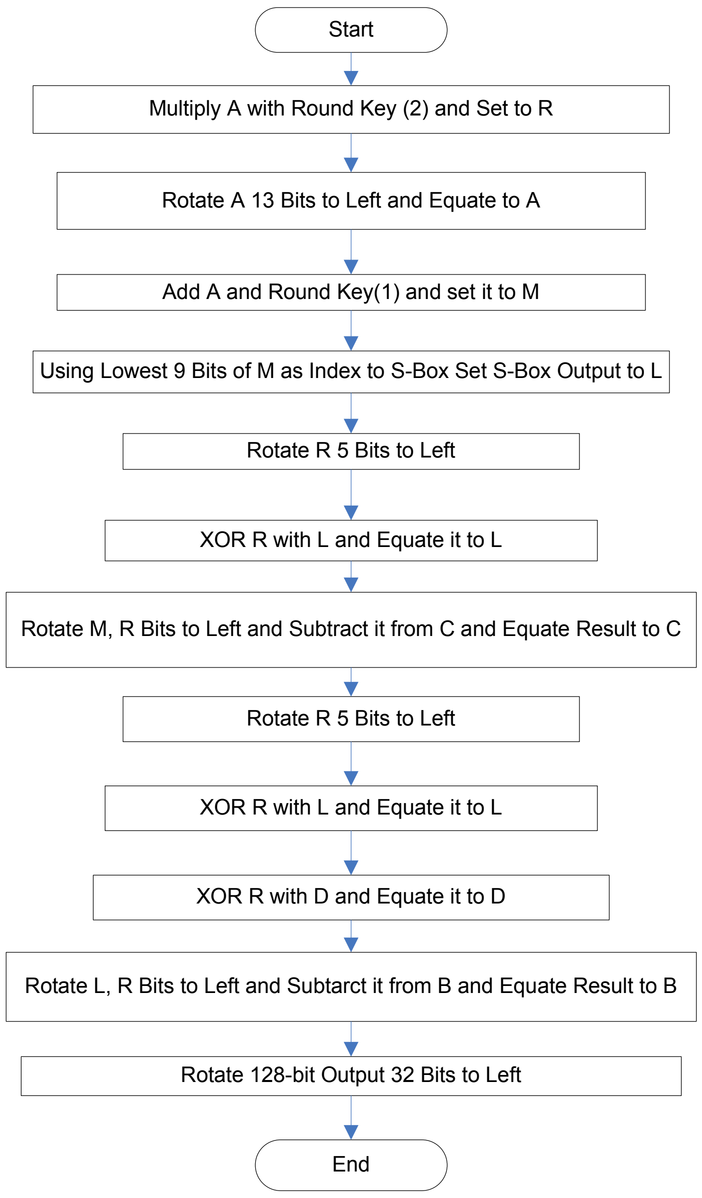

Start
Multiply A with Round Key (2) and Set to R
Rotate A 13 Bits to Left and Equate to A
Add A and Round Key(1) and set it to M
Using Lowest 9 Bits of M as Index to S-Box Set S-Box Output to L
Rotate R 5 Bits to Left
XOR R with L and Equate it to L
Rotate M, R Bits to Left and Subtract it from C and Equate Result to C
Rotate R 5 Bits to Left
XOR R with L and Equate it to L
XOR R with D and Equate it to D
Rotate L, R Bits to Left and Subtarct it from B and Equate Result to B
Rotate 128-bit Output 32 Bits to Left
End

Figure A.7. Flowchart of Reverse Keyed Transform

A.2 FLOWCHARTS OF RC6

A.2.1 Flowchart of Encryption

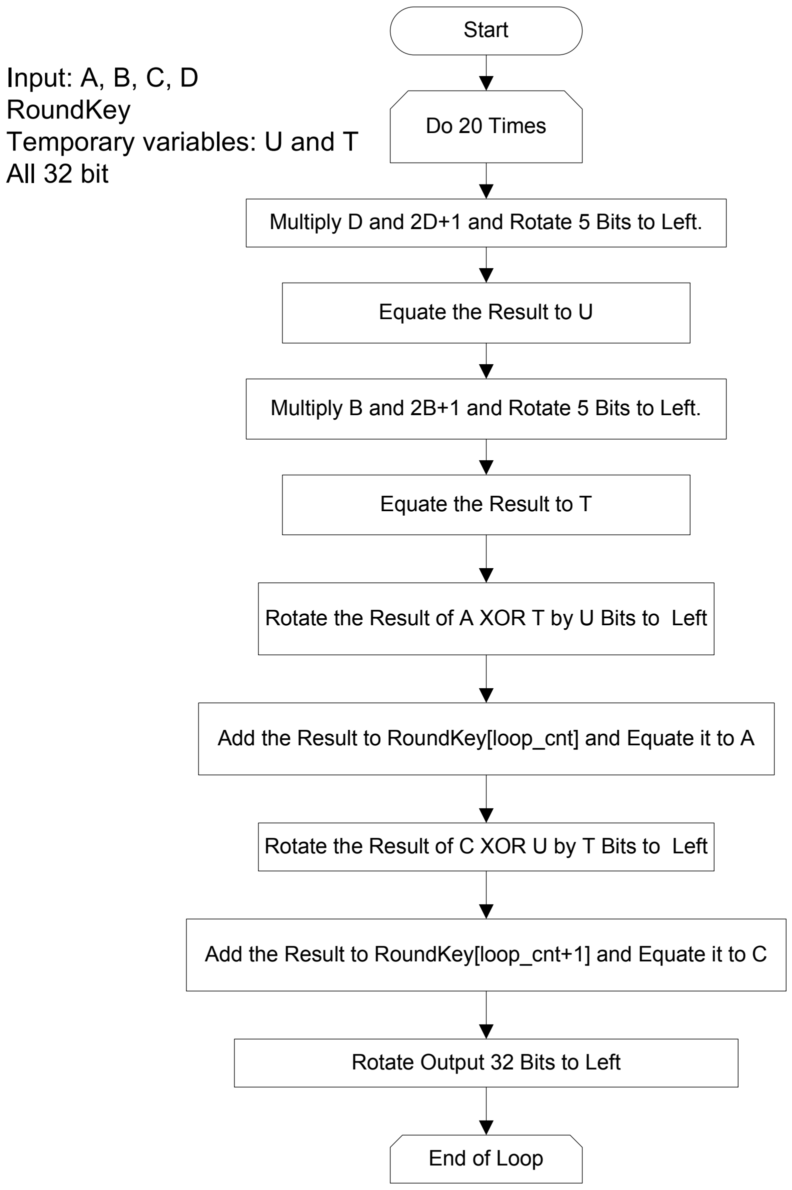

Start
Input: A, B, C, D
RoundKey
Temporary variables: U and T
All 32 bit
Do 20 Times
Multiply D and 2D+1 and Rotate 5 Bits to Left.
Equate the Result to U
Multiply B and 2B+1 and Rotate 5 Bits to Left.
Equate the Result to T
Rotate the Result of A XOR T by U Bits to Left
Add the Result to RoundKey[loop_cnt] and Equate it to A
Rotate the Result of C XOR U by T Bits to Left
Add the Result to RoundKey[loop_cnt+1] and Equate it to C
Rotate Output 32 Bits to Left
End of Loop

Figure A.8. Flowchart of encryptasm of RC6

A.2.2 Flowchart of Decryption

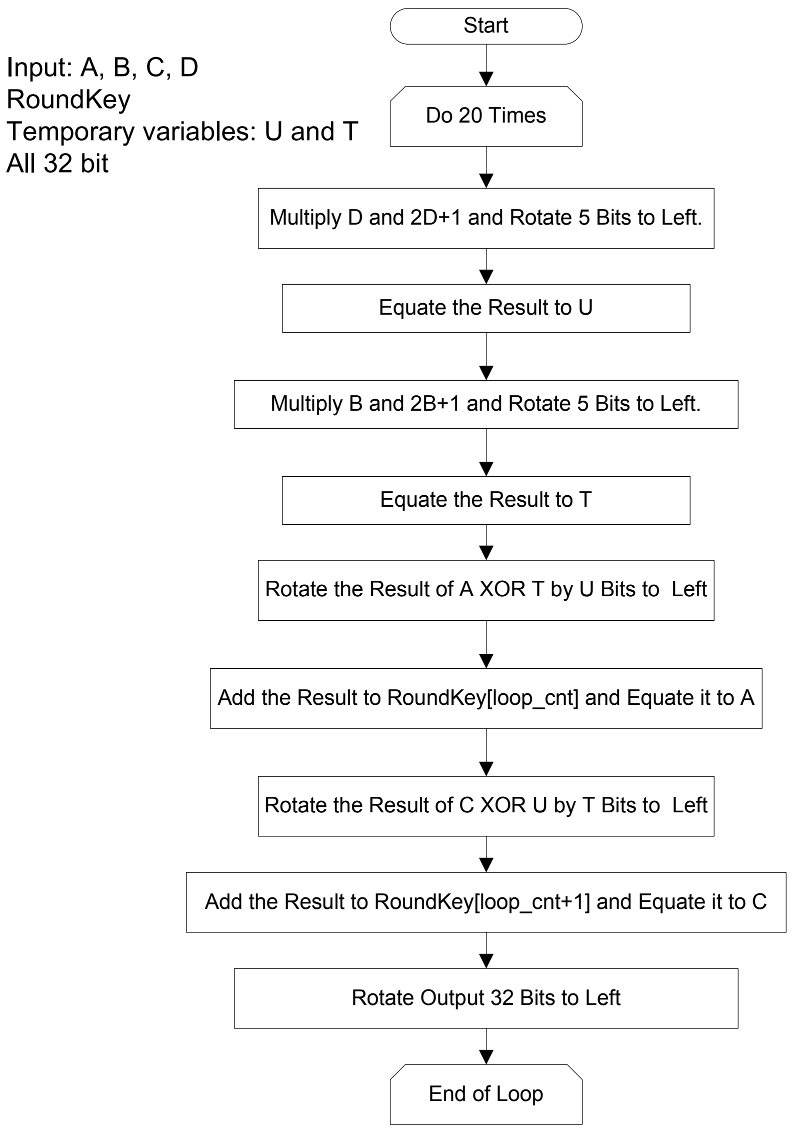

Input: A, B, C, D
RoundKey
Temporary variables: U and T
All 32 bit
Start
Do 20 Times
Multiply D and 2D+1 and Rotate 5 Bits to Left.
Equate the Result to U
Multiply B and 2B+1 and Rotate 5 Bits to Left.
Equate the Result to T
Rotate the Result of A XOR T by U Bits to Left
Add the Result to RoundKey[loop_cnt] and Equate it to A
Rotate the Result of C XOR U by T Bits to Left
Add the Result to RoundKey[loop_cnt+1] and Equate it to C
Rotate Output 32 Bits to Left
End of Loop

Figure A.9. Flowchart of decryptasm of RC6

A.3 FLOWCHARTS OF RIJNDAEL

A.3.1 Flowchart of Encryption

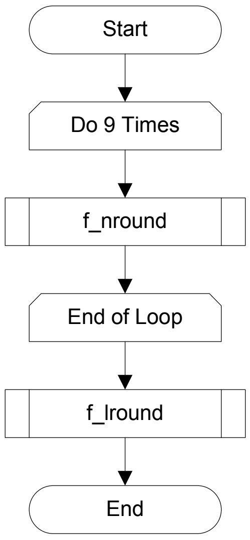


Figure A.10. Flowchart of encryptasm of RIJNDAEL

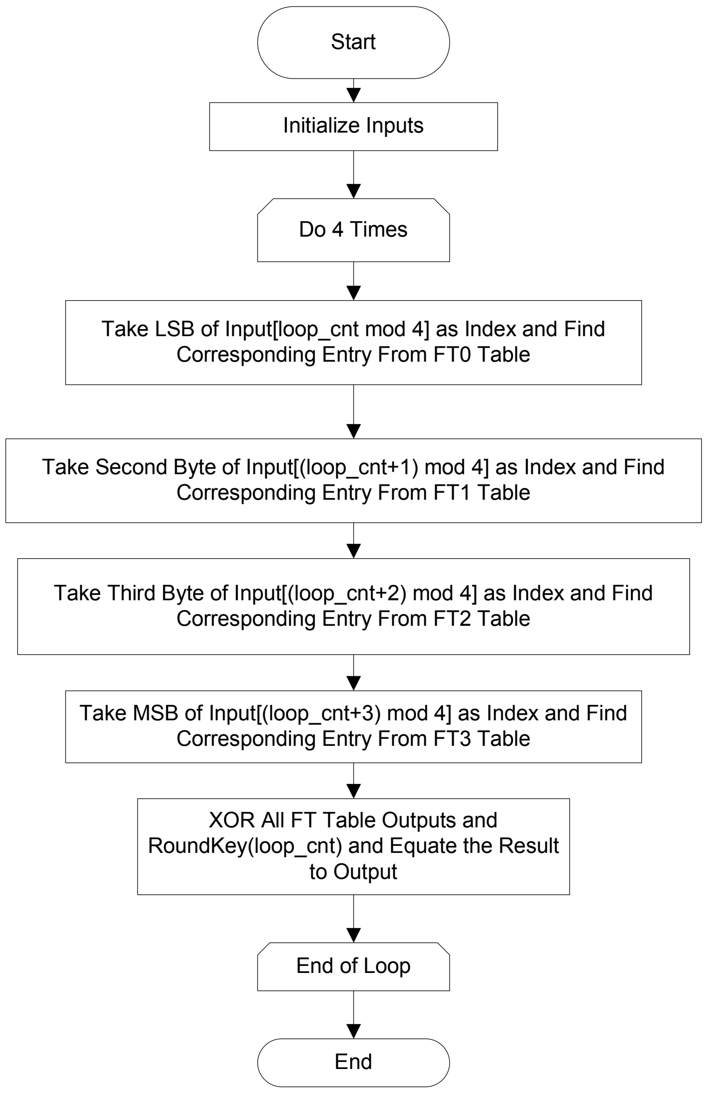

Start
Initialize Inputs
Do 4 Times
Take LSB of Input[loop_cnt mod 4] as Index and Find Corresponding Entry From FT0 Table
Take Second Byte of Input[(loop_cnt+1) mod 4] as Index and Find Corresponding Entry From FT1 Table
Take Third Byte of Input[(loop_cnt+2) mod 4] as Index and Find Corresponding Entry From FT2 Table
Take MSB of Input[(loop_cnt+3) mod 4] as Index and Find Corresponding Entry From FT3 Table
XOR All FT Table Outputs and RoundKey(loop_cnt) and Equate the Result to Output
End of Loop
End

Figure A.11. Flowchart of f_nround

A.3.2 Flowchart of Decryption

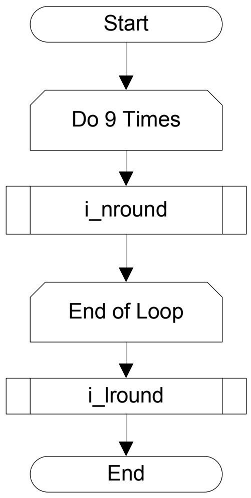


Figure A.12. Flowchart of decryptasm of RIJNDAEL

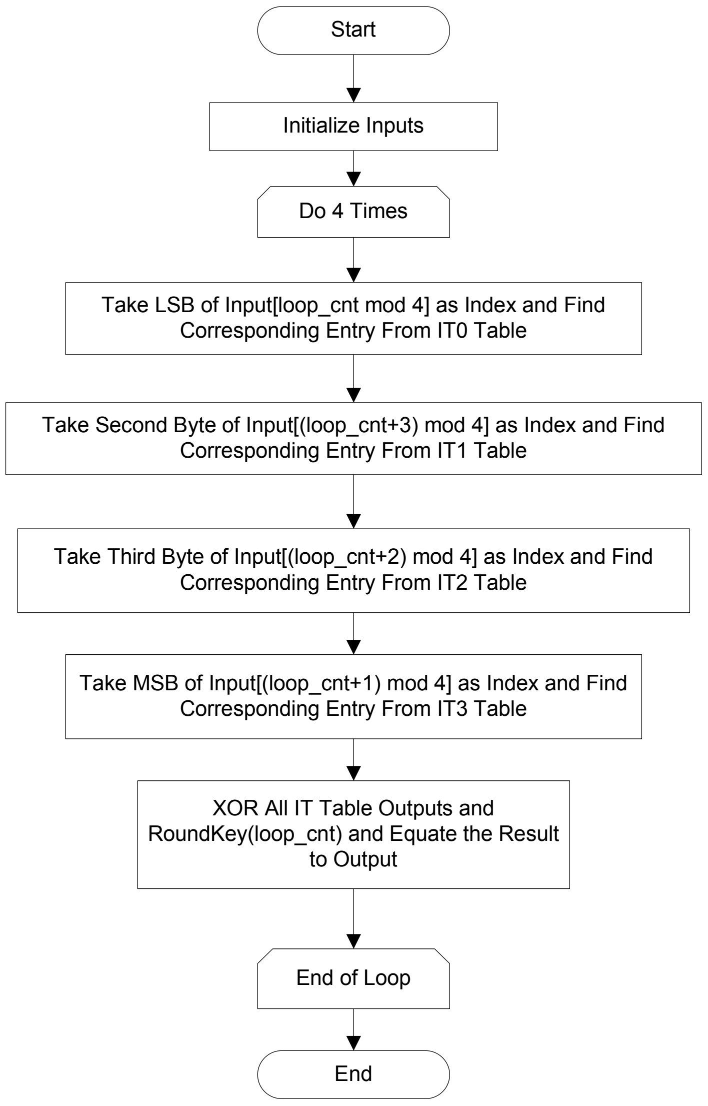

Start
Initialize Inputs
Do 4 Times
Take LSB of Input[loop_cnt mod 4] as Index and Find Corresponding Entry From IT0 Table
Take Second Byte of Input[(loop_cnt+3) mod 4] as Index and Find Corresponding Entry From IT1 Table
Take Third Byte of Input[(loop_cnt+2) mod 4] as Index and Find Corresponding Entry From IT2 Table
Take MSB of Input[(loop_cnt+1) mod 4] as Index and Find Corresponding Entry From IT3 Table
XOR All IT Table Outputs and RoundKey(loop_cnt) and Equate the Result to Output
End of Loop
End

Figure A.13. Flowchart of i_nround

APPENDIX B

ASSEMBLER CODE PARTS OF RC6 AND RIJNDAEL

B.1 MULTIPLICATION USED FOR RC6

LD *AR2, T

Load the data which ar2 points, which is the low 16 bits of first multiplier, to the T, temporary register

MPYU *AR5-, A

Multiply the low bits of first multiplier with second multiplier and store the result in accumulator A

STL A, u_low

Store low 16bits of the result to location u_low

LD A,-16, A

Shift the result 16 bits to right and load it to accumulator A

MACSU *AR2-,*AR5+, A

Multiply high 16 bits of second multiplier with low 16 bits of first multiplier and add the result to accumulator A

MACSU *AR5-,*AR2+, A

Multiply low 16 bits of second multiplier with high 16 bits of first multiplier and add the result to accumulator A

STL A, u_high

Store low 16 bits of result to location u_high

B.2 TABLE LOOKUP FOR RIJNDAEL

LD ***AR2+%, A**

Load the LSW of first input to accumulator A

MAR ***AR2+%**

Prepare pointer for getting second byte

AND **NO_FF, A**

Take LSB of the input

SFTL **A, 1**

Multiply the index by two by shifting 1 bit left

ADD **PTR_FT_TAB0, A**

Add the table 0 start pointer to accumulator A

STLM **A, AR3**

Store the table 0 index to AR3

LD ***AR2+%, A**

Load the LSW of second input to accumulator A

SFTL **A,-8**

Take second byte of the input

SFTL **A, 1**

Multiply the index by two by shifting 1 bit left

ADD **PTR_FT_TAB1, A**

Add the table 1 start pointer to accumulator A

STLM **A, AR4**

Store the table 1 index to AR4

```
LD        *AR2+%, A
```

Load the MSW of third input to accumulator A

```
MAR       *AR2+%
```

Prepare pointer for getting fourth byte

```
AND       NO_FF,A
```

Take LSB of the third input

```
SFTL      A, 1
```

Multiply the index by two by shifting 1 bit left

```
ADD       PTR_FT_TAB2, A
```

Add the table 2 start pointer to accumulator A

```
STLM      A, AR5
```

Store the table 2 index to AR5

```
LD        *AR2+0%, A
```

Load the MSW of fourth input to accumulator A and do modulo 4 operation

```
SFTL      A,-8
```

Take MSB of the input

```
SFTL       A, 1
```

Multiply the index by two by shifting 1 bit left

```
ADD       PTR_FT_TAB3, A
```

Add the table 3 start pointer to accumulator A

```
STLM      A, AR6
```

Store the table 2 index to AR6

```
DLD       *AR3, A
DLD       *AR4, B
XOR       A, B
DLD       *AR5, A
XOR        A, B
```

```
DLD         *AR6, A
XOR          A, B
DLD         *AR7+, A
XOR          A, B
DST         B, *AR1+
```

XOR all table outputs with the key and store the result in AR1

APPENDIX C

EFFICIENT IMPLEMENTATION PROCEDURE FOR RIJNDAEL

(Taken from [Daemen, Rijmen, 1999] pages 17-18)Express one column of the round output e in terms of bytes of the round input a. $a_{i,j}$ denotes the byte of a in row i and column j, a_j denotes the column j of State a. For the key addition and the MixColumn transformation, we have

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = \begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix} \text{ and } \begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix}. \tag{C.1}$$

For the ShiftRow and the ByteSub transformations, we have:

$$\begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix} = \begin{bmatrix} b_{0,j} \\ b_{1,j-C1} \\ b_{2,j-C2} \\ b_{3,j-C3} \end{bmatrix} \text{ and } b_{i,j} = \mathrm{S}\left[a_{i,j}\right]. \tag{C.2}$$

In this expression, the column indices must be taken modulo **4**. By substitution, the above expressions can be combined into:

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} \mathrm{S}\left[a_{0,j}\right] \\ \mathrm{S}\left[a_{1,j-C1}\right] \\ \mathrm{S}\left[a_{2,j-C2}\right] \\ \mathrm{S}\left[a_{3,j-C3}\right] \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}. \tag{C.3}$$

The matrix multiplication can be expressed as a linear combination of vectors:

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = \mathrm{S}\left[a_{0,j}\right]\begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} \oplus \mathrm{S}\left[a_{1,j-C1}\right]\begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} \oplus \mathrm{S}\left[a_{2,j-C2}\right]\begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} \oplus \mathrm{S}\left[a_{3,j-C3}\right]\begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}. \qquad \text{(C.4)}$$

The multiplication factors S[$a_{i,j}$] of the four vectors are obtained by performing a table lookup on input bytes $a_{i,j}$ in the S-box table S[256].

We define tables T_0 to T_3 :

$$\mathrm{T}_0[a] = \begin{bmatrix} \mathrm{S}[a]\bullet 02 \\ \mathrm{S}[a] \\ \mathrm{S}[a] \\ \mathrm{S}[a]\bullet 03 \end{bmatrix} \quad \mathrm{T}_1[a] = \begin{bmatrix} \mathrm{S}[a]\bullet 03 \\ \mathrm{S}[a]\bullet 02 \\ \mathrm{S}[a] \\ \mathrm{S}[a] \end{bmatrix} \quad T_2[a] = \begin{bmatrix} \mathrm{S}[a] \\ \mathrm{S}[a]\bullet 03 \\ \mathrm{S}[a]\bullet 02 \\ \mathrm{S}[a] \end{bmatrix} \quad \mathrm{T}_3[a] = \begin{bmatrix} \mathrm{S}[a] \\ \mathrm{S}[a] \\ \mathrm{S}[a]\bullet 03 \\ \mathrm{S}[a]\bullet 02 \end{bmatrix}. \qquad \text{(C.5)}$$

These are 4 tables with 256 4-byte word entries and make up for 4KByte of total space. Using these tables, the round transformation can be expressed as:

$$e_j = \mathrm{T}_0\left[a_{0,j}\right] \oplus \mathrm{T}_1\left[a_{1,j-C1}\right] \oplus \mathrm{T}_2\left[a_{2,j-C2}\right] \oplus \mathrm{T}_3\left[a_{3,j-C3}\right] \oplus k_j. \qquad \text{(C.6)}$$

In our implementation, T_0, T_1, T_2, T_3 are defined as ft_table0, ft_table1,ft_table2 and ft_table3. In a similar way, we define inverse transform tables for decryption and apply the same procedure.