A STUDY ON PARALLEL IMPLEMENTATION OF ADVANCED ENCRYPTION STANDARD (AES)

by

Sabbir Mahmud

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science
in
Computer Science

INDEPENDENT UNIVERSITY, BANGLADESH

May 27, 2004

A STUDY ON PARALLEL IMPLEMENTATION OF
ADVANCED ENCRYPTION STANDARD (AES)

by

Sabbir Mahmud

has been approved

June 2004

APPROVED:

Prof. Dr. M. Lutfar Rahman , Chairperson

Dr. Indrani Haque , Member

Dr. Khosru Salim , Member

Dr. Mostofa Akbar , Member

Supervisory Committee

ACCEPTED:

_____
Director, School of Communication

ABSTRACT

Most cryptographic algorithms function more efficiently when implemented in hardware than in software running on single processor. However, systems that use hardware implementations have significant drawbacks: they are unable to respond to flaws discovered in the implemented algorithm or to changes in standards. As an alternative, it is possible to implement cryptographic algorithms in software running on multiple processors. However, most of the cryptographic algorithms like DES (Data Encryption Standard) or 3DES have some drawbacks when implemented in software: DES is no longer secure as computers get more powerful while 3DES is relatively sluggish in software. AES (Advanced Encryption Standard), which is rapidly being adopted worldwide, provides a better combination of performance and enhanced network security than DES or 3DES by being computationally more efficient than these earlier standards. Furthermore, by supporting large key sizes of 128, 192, and 256 bits, AES offers higher security against brute-force attacks.

In this thesis, AES has been implemented with single processor. Then the result has been compared with parallel implementations of AES with 2, 4, and 8 processors varying different parameters such as key size, number of rounds and extended key size, and show how parallel implementation of the AES offers better performance yet flexible enough for cryptographic algorithms.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Information Security and Cryptography

The emergence of the Internet as a trusted medium for commerce and communication has made cryptography an essential component of modern information systems. Cryptography provides the mechanisms necessary to implement accountability, accuracy, and confidentiality in communications [1]. As demands for secure communication bandwidth grow, efficient cryptographic processing will become increasingly vital to good system performance. To introduce cryptography, an understanding of issues related to information security in general is necessary. Information security manifests itself in many ways according to the situation and requirement. Regardless of who is involved, to one degree or another, all parties to a transaction must have confidence that certain objectives associated with information security have been met.

Over the centuries, an elaborate set of protocols and mechanisms has been created to deal with information security issues when the information is conveyed by physical documents. Often the objectives of information security cannot solely be achieved through mathematical algorithms and protocols alone, but require procedural techniques and abidance of laws to achieve the desired result. For example, privacy of letters is provided by sealed envelopes delivered by an accepted mail service. The physical security of the envelope is, for practical necessity, limited and so laws are enacted which make it a criminal offense to open mail for which one is not authorized. It is sometimes the case that security is achieved not through the information itself but through the physical document recording it. For example, paper currency requires special inks and materials to prevent counterfeiting.

Conceptually, the way information is recorded has not changed dramatically over time. Whereas information was typically stored and transmitted on paper, much of it now resides on magnetic media and is transmitted via telecommunications systems. What has changed dramatically is the ability to copy and alter information. One can make thousands of identical copies of a piece of information stored electronically and each is indistinguishable from the original. With information on paper, this is much more difficult. What is needed then for a society where information is mostly stored and transmitted in electronic form is a means to ensure information security which is independent of the physical medium recording or conveying it and such that the objectives of information security rely solely on digital information itself.

One of the fundamental tools used in information security is the signature. It is a building block for many other services such as non-repudiation, data origin authentication, identification, and witnessing, to mention a few. Having learned the basics in writing, an individual is taught how to produce a handwritten signature for the purpose of identification.

At contract age the signature evolves to take on a very integral part of the person's identity. This signature is intended to be unique to the individual and serve as a means to identify, authorize, and validate. With electronic information the concept of a signature needs to be redressed; it cannot simply be something unique to the signer and independent of the information signed. Electronic replication of it is so simple that appending a signature to a document not signed by the originator of the signature is almost a triviality.

Achieving information security in an electronic society requires a vast array of technical and legal skills. There is, however, no guarantee that all of the information security objectives deemed necessary can be adequately met. The technical means is provided through **cryptography**.

Cryptography is the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication, and data origin authentication [1]. Cryptography is not the only means of providing information security, but rather one set of techniques.

Cryptographic goals

The following four cryptographic goals form a framework from which other goals are derived:

Confidentiality is a service used to keep the content of information from all but those authorized to have it.

Data integrity is a service which addresses the unauthorized alteration of data. 3.

Authentication is a service related to identification.

Non-repudiation is a service which prevents an entity from denying previous commitments or actions. When disputes arise due to an entity denying that certain actions were taken, a means to resolve the situation is necessary.

A fundamental goal of cryptography is to adequately address these four areas in both theory and practice. Cryptography is about the prevention and detection of cheating and other malicious activities.

Cryptography, over the ages, has been an art practiced by many who have devised ad hoc techniques to meet some of the information security requirements. The last twenty years have been a period of transition as the discipline moved from an art to a science. There are now several international scientific conferences devoted exclusively to cryptography and also an international scientific organization, the International Association for Cryptologic Research (IACR), aimed at fostering research in the area.

1.2 Symmetric-key Cryptography and AES

Symmetric-key cryptography, also called secret key cryptography, is the most intuitive kind of cryptography. It involves the use of a secret key known only to the participants of the secure communication. Symmetric-key cryptography can be used to transmit information over an insecure channel, but it has also other uses, such as secure storage on insecure media or strong mutual authentication. In symmetric-key cryptography, the key must be shared by both the sender and the receiver. The sender applies the encryption function using the key to the plaintext to produce the ciphertext. The ciphertext is sent to the receiver, who then applies the decryption function using the same shared key. Since the plaintext cannot be derived from the ciphertext without knowledge of the key, the ciphertext can be sent over public networks such as the Internet. Therefore, symmetric key cryptography is characterized by the use of a single key to perform both the encrypting and decrypting of data. Since the algorithms are public knowledge, security is determined by the level of protection afforded the key. If key is kept secret, both the secrecy and authentication services are provided. Secrecy is provided, because if the message is intercepted, the intruder cannot transform the ciphertext into its plaintext format. Assuming that only two users know the key, authentication is provided because only a user with the key can generate ciphertext that a recipient can transform into meaningful plaintext.

The United States standard for Symmetric-key cryptography, in which the same key is used for both encryption and decryption, is the **Data Encryption Standard** (DES) [2]. This is based upon a combination and permutation of shifts and exclusive OR operations and so can be very fast when implemented directly on hardware (1 GByte/s throughput or better) or on general purpose processors. DES uses a 56-bit key and maps a 64-bit input block of plaintext onto a 64-bit output block of ciphertext. 56 bits is a rather small key for today's computing power, the key size is

indeed one of the most controversial aspects of this algorithm. The mainstream cryptographic community has long held that DES's 56-bit key is too short to withstand a brute-force attack from modern computers [3]. Computers have made it easier to attack ciphertext by using brute force methods rather than by attacking the mathematics. With a brute force attack, the attacker merely generates every possible key and applies it to the ciphertext. Any resulting plaintext that makes sense offers a candidate for a legitimate key.

The current key size of 56 bits (plus 8 parity bits) of DES is now starting to seem small, but the use of larger keys with triple DES (3DES) can generate much greater security. If security is the only consideration, then 3DES will be an appropriate choice for a standardized encryption algorithm for decades to come. However, the principle drawback of 3DES is that the algorithm is relatively sluggish in software. The original DES was designed for mid 1970s hardware implementation and does not produce efficient software code. 3DES, which has three times as many rounds as DES, is correspondingly slower. A secondary drawback is that both DES and 3DES use a 64-bit block size. For reasons of both efficiency and security, a larger block size is desirable.

Because of these drawbacks, 3DES is not a reasonable candidate for long term use. As a replacement, NIST (National Institute of Standards and Technology) of USA in 1997 issued a call for proposals for a new **Advanced Encryption Standard (AES)**, which would have security length equal to or better than 3DES and significantly, improved efficiency. In addition to these general requirements, NIST specified that AES must be a symmetric block cipher with a block length of 128 bits and support for key lengths of 128, 192, and 256 bits. The AES algorithm was selected in October 2001 after a multi-year evaluation process led by NIST with submissions and review by an international community of cryptography experts and the Rijndael algorithm [4], invented by Joan Daemen and Vincent Rijmen, was selected as the standard

which was published in November 2002. NIST's intent was to have a cipher that will remain secure well into the next century.

1.3 Aims of the Thesis

The main aims of this thesis are to implement AES in serial, find out its performance on single processor and compare the performance with parallel implementations of AES with 2, 4, and 8 processors varying different parameters such as key size, number of rounds and extended key size, and show how parallel implementation of the AES offers better performance yet flexible enough for cryptographic algorithms.

1.4 Layout of the Thesis

This thesis is divided into seven major parts. This chapter introduces the relationship between the information security and cryptography, cryptographic goals, concepts of symmetric-key cryptography, the most secured symmetric-key cryptographic algorithm: Advanced Encryption Standard (AES) and the aims of the thesis.

Chapter 2 gives a concise introduction to the basic building blocks of all encryption techniques: substitution and transposition techniques. A detailed discussion on symmetric–key and public-key cryptography, the importance of symmetric–key cryptography, limitations of DES and reason for using AES is discussed in this chapter. **Chapter 3** describes the overall structure and the working principle of AES. **Chapter 4** introduces the mathematical background of AES algorithm. The Finite Fields and its different operations in the form of GF (2^8) are described in details before describing the algorithm itself. **Chapter 5** introduces different types of parallel architectures and interconnection networks before the parallel algorithm of AES is given. Some sample input/output are shown varying the key size, number of rounds and the number of processors to verify the correctness of parallel algorithm. Finally, the run time complexity of the parallel algorithm is shown to measure the

performance improvement of the parallel implementation over the serial implementation.

Chapter 6 summarizes the total work along with the opportunities for the future work. Appendix A and B present the program listings for both serial and parallel implementations of AES.

CHAPTER 2

CONVENTIONAL ENCRYPTION AND AES

2.1 Introduction

Cryptography (from Greek *kryptÃ³s*, "hidden", and *grÃiphein*, "to write") is generally understood to be the study of the principles and techniques by which information is converted into an encrypted version that is difficult (ideally impossible) for any unauthorized person to convert to the original information, while still allowing the intended reader to do so. In fact, cryptography covers rather more than merely encryption and decryption. It is, in practice, a specialized branch of information theory with substantial additions from other branches of mathematics. Cryptography is probably the most important aspect of communications security and is becoming increasingly important as a basic building block for computer security.

The increased use of computer and communications systems by the industry has increased the risk of theft of proprietary information. Although these threats may require a variety of countermeasures, cryptography is a primary method of protecting valuable electronic information. In data and telecommunications, cryptography is necessary when communicating over any unsecured medium, which includes just about any network, particularly the Internet. Within the context of any application-to-application communication, there are some specific security requirements, including:

Authentication*:* The process of proving one's identity.

Confidentiality*:* Ensuring that no one can read the message except the intended receiver.

Integrity*:* Assuring the receiver that the received message has not been altered in any way from the original.

Non-repudiation*:* A mechanism to prove that the sender really sent this message.

There are, in general, two types of cryptographic schemes typically used to accomplish these goals: **secret key** (or **symmetric** or **conventional**) cryptography and **public-key** (or asymmetric) cryptography. In **symmetric-key** cryptography, an algorithm is used to scramble the message using a secret key in such a way that it becomes unusable to all except the ones that have access to that secret key. The most widely known symmetric cryptographic algorithm is DES, developed by IBM in the seventies. It uses a key of 56 bits and operates on chunks of 64 bits at a time. In **public key** cryptography [4], algorithms use two different keys: a **private** and a **public** one. A message encrypted with a private key can be decrypted with its public key (and vice versa). The owner of the key pair holds the private key, and may distribute the public key to anyone. Someone who wants to send a secret message uses the public key of the intended receiver to encrypt it. Only the receiver holds the private key and can decrypt it.

The two basic building blocks of all encryption techniques are **substitution** and **transposition**. A substitution technique is one in which the letters of plaintext are replaced by other letters or by numbers or symbols. Transposition technique is a different kind of mapping where mapping is achieved by performing some sort of permutation on the plaintext letter.

In this chapter, substitution and transposition techniques will be discussed, followed by a comprehensive discussion on symmetric–key and public-key cryptography. Finally, the importance of symmetric–key cryptography, limitations of DES and reason for using AES (Advanced Encryption Standard) will be discussed.

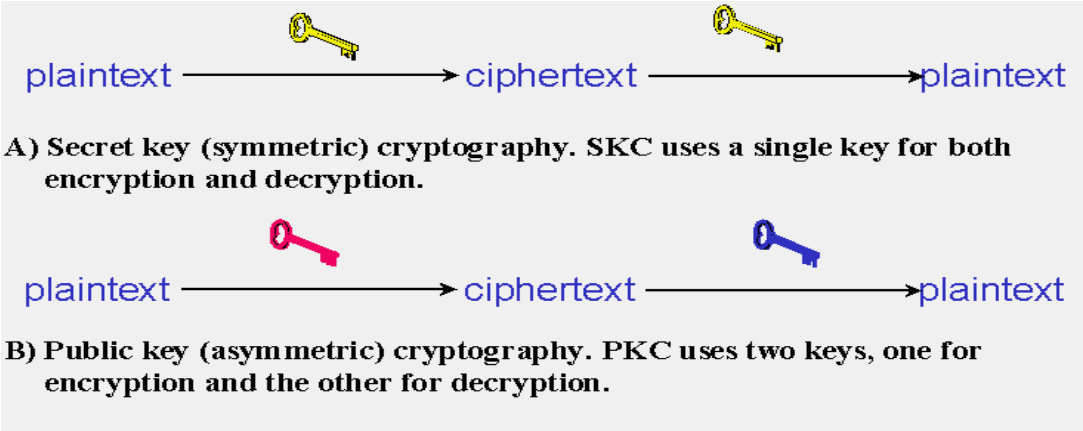


Figure 2.1 Two types of cryptography

2.2 Substitution Techniques

2.2.1 Monoalphabetic Cipher:

A simple substitution cipher is called monoalphabetic because there is one-to-one correspondence between a character and its substitute. In other words, a monoalphabetic cipher is one which replaces a character in the plaintext by another character. Thus, simple one-to-one mapping from plaintext to ciphertext character is used to encipher an entire message. There are 26! = 4 X 10^{26} different monoalphabetic ciphers corresponding to the permutations of the English alphabet. The key is permuted alphabet. An example of a monoalphabetic substitution is shown below:

PLAINTEXT: abcdefghijklmnopqrstuvwxyz

CIPHERTEXT: QRSKOWEIPLTUYACZMNVDHFGXJB

2.2.2 Caesar Cipher

One of the simplest examples of a substitution cipher is the **Caesar cipher**, which is said to have been used by Julius Caesar to communicate with his army. Caesar is considered to be one of the first persons to have ever employed encryption for the sake of securing messages. Caesar decided that shifting each letter in the message

would be his standard algorithm, and so he informed all of his generals of his decision, and was then able to send them secured messages. Using the Caesar Shift (3 to the right), the message,

"RETURN TO ROME"

would be encrypted as,

"UHWXUA WR URPH"

In this example, 'R' is shifted to 'U', 'E' is shifted to 'H', and so on. Thus, the Caesar cipher is a *shift* cipher since the ciphertext alphabet is derived from the plaintext alphabet by shifting each letter a certain number of spaces.

2.2.3 Playfair Cipher

The best known substitution cipher that encrypts pairs of letters is the **Playfair Cipher** invented by **Sir Charles Wheatstone** [5] but championed at the British Foreign Office by Lyon Playfair, the first Baron Playfair of St. Andrews, whose name the cipher bears. The cipher is a form of monoalphabetic substitution, but relies on **DIGRAPHS** rather than single letters. The playfair cipher is believed to be the first digraphic system. Here, a 5 x 5-square matrix containing the 26 letters of the alphabet (I and J are treated as the same letter) is used to carry out the encryption. A key word, **MONARCHY** in this example, is filled in first, and the remaining unused letters of the alphabet are entered in their lexicographic order:

M	**O**	**N**	**A**	**R**
C	**H**	**Y**	B	D
E	F	G	I/J	K
L	P	Q	S	T
U	V	W	X	Z

Pairs of plaintext letters are encrypted with the matrix by first locating the two plaintext letters in the matrix. They are

- in different rows and columns, or
- in the same row, or
- in the same column, or
- alike.

The corresponding encryption (replacement) rules are the following:

1. If the pair of letters are in different rows and columns, each letter is replaced by the letter that is in the same row but in the other column; *i.e.,* to encrypt WE, W is replaced by U and E by G.
2. If two letters are in the same row simply shift both one position to the right i.e. A and R are in the same row. A is encrypted as R and R (reading the row cyclically) as M.
3. Similarly, if two letters are in the same column shift both one position down i.e. I and S are in the same column. I is encrypted as S and S as X.
4. If a double letter occurs, a spurious symbol, say Q, is introduced so that the MM in SUMMER would encrypt into NL for MQ and CL for ME.
5. An X is appended to the end of the plaintext if necessary to cause the plaintext to have an even number of letters.

Encrypting the following plaintext using Sayer's Playfair matrix yields:

PlainText:	**WE**	**ARE**	**DISCOVERED**	**SAVE**	**YOURSELFX**
Ciphertext:	**UG**	**RMK**	**CSXHMUFMKBTOXG**		**CMVATLUIV**

Breaking some substitution ciphers with the aid of letter frequencies:

The Caesar Cipher, Playfair Cipher and the Monoalphabetic Ciphers have one property in common: **"Same plain letters are encoded to the same cipher letter**." i.e. in the Caesar Cipher each "a" turned into "d", each "b" turned into "e", etc. The reason why such Ciphers can be broken is the following:

Although letters are changed the underlying letter frequencies are not! If the plain letter "a" occurs 10 times its cipher letter will do so 10 times. Therefore, some substitution cipher can be broken with the aid of letter frequency analysis**.** The most difficult substitution cipher to break is the one where each plain letter is randomly assigned to its cipher letter. Instead of using encryption functions, tables are used below to describe plain-cipher letter correspondences i.e. each (capital) plain letter can be assigned to a cipher letter as follows:

A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
q	**a**	**z**	**w**	**s**	**x**	**e**	**d**	**c**	**r**	**F**	**v**	**t**	**g**	**b**	**y**	**h**	**n**	**u**	**j**	**m**	**i**	**k**	**o**	**l**	**p**

"Brute force" - attacks to break the cipher are hopeless since there are 26! = 403291461126605635584000000 or about $4 * 10^{26}$ many possible ways to encode the 26 letters of the English alphabet. In order to crack the random substitution cipher, however, one advantage of the fact can be taken into to the consideration that the underlying letter frequencies of the original plain text don't get lost. This fact makes the Random Substitution Cipher very susceptible to cipher attacks. An eavesdropper literally just needs to count the letter frequencies of the cipher letters. Moreover, the most frequent letters in the English language are ETNORIA which – except for the O - occur even as the most frequent letters in the brief virus carrier message. And the longer the messages are the more do the relative frequencies of the cipher letters approach the expected frequencies.

2.2.4 Polyalphabetic Cipher

One of the main problems with simple substitution ciphers is that they are so vulnerable to frequency analysis. Given a sufficiently large ciphertext, it can easily be broken by mapping the frequency of its letters to the known frequencies of, say, English text. Therefore, to make ciphers more secure, cryptographers have long been interested in developing enciphering techniques that are immune to frequency

analysis. One of the most common approaches is to suppress the normal frequency data by using more than one alphabet to encrypt the message. A **polyalphabetic substitution** [6] cipher involves the use of two or more cipher alphabets. Instead of there being a one-to-one relationship between each letter and its substitute, there is a one-to-many relationship between each letter and its substitutes.

The Vigenere Tableau

The **polyalphabetic Cipher***,* proposed by Blaise de Vigenere from the court of Henry III of France in the sixteenth century, is a polyalphabetic substitution based on the following *tableau:*

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

The polyalphabetic cipher uses this table together with a keyword to encipher a message. For example, to encipher the following plaintext message:

TO BE OR NOT TO BE THAT IS THE QUESTION

using the keyword **RELATIONS**. Encryption begins by writing the keyword, repeated as many times as necessary, above the plaintext message. To derive the ciphertext using the tableau, for each letter in the plaintext, by finding the intersection of the row given by the corresponding keyword letter and the column given by the plaintext letter itself to pick out the ciphertext letter.

Keyword:	**RELAT**	**IONSR**	**ELATI**	**ONSRE**	**LATIO**	**NSREL**
Plaintext:	**TOBEO**	**RNOTT**	**OBETH**	**ATIST**	**HEQUE**	**STION**
Ciphertext:	**KSMEH**	**ZBBLK**	**SMEMP**	**OGAJX**	**SEJCS**	**FLZSY**

Decipherment of an encrypted message is equally straightforward. One writes the keyword repeatedly above the message:

Keyword:	**RELAT**	**IONSR**	**ELATI**	**ONSRE**	**LATIO**	**NSREL**
Ciphertext:	**KSMEH**	**ZBBLK**	**SMEMP**	**OGAJX**	**SEJCS**	**FLZSY**
Plaintext:	**TOBEO**	**RNOTT**	**OBETH**	**ATIST**	**HEQUE**	**STION**

Decryption process uses the keyword letter to pick a column of the table and then traces down the column to the row containing the ciphertext letter. The index of that row is the plaintext letter.

The strength of the Vigenere cipher against frequency analysis can be seen by examining the above ciphertext. There are 7 'T's in the plaintext message and that they have been encrypted by 'H,' 'L,' 'K,' 'M,' 'G,' 'X,' and 'L' respectively. This successfully masks the frequency characteristics of the English 'T.' One way of looking at this is to notice that each letter of the keyword RELATIONS picks out 1 of the 26 possible substitution alphabets given in the Vigenere tableau. Thus, any message encrypted by a Vigenere cipher is a collection of as many simple substitution ciphers as there are letters in the keyword.

2.3 Transposition Techniques

In transposition techniques plain letters are simply rearranged**. Rail Fence Cipher** was used by the Spartans to send messages to Greek warriors. Here, the plaintext is staggered between rows and the rows are then read sequentially to give the cipher. The **key** is the number of rows used to encode. For example, a rail fence with two rows turns the message

"**transposition ciphers can easily be broken**" into

t	**a**	**s**	**o**	**i**	**i**	**n**	**i**	**H**	**r**	**c**	**n**	**A**	**i**	**y**	**e**	**r**	**k**	**n**
r	**n**	**p**	**s**	**t**	**o**	**c**	**p**	**E**	**s**	**a**	**e**	**S**	**l**	**b**	**b**	**o**	**e**	

and creates the cipher text

"**tasoiinihrcnaiyerknrnpstocpesaeslbboe"**

Using Key = 3, the same plain text can be converted into different ciphertext

T	**n**	**o**	**t**	**n**	**p**	**r**	**a**	**a**	**l**	**e**	**o**	**N**
R	**s**	**s**	**i**	**c**	**h**	**s**	**n**	**s**	**y**	**b**	**k**	
a	**p**	**i**	**O**	**I**	**e**	**c**	**e**	**i**	**b**	**r**	**e**	

Ciphertext = tnotnpraaleonrssichsnsybkapioieceibre

The Rail Fence is the simplest example of a class of transposition ciphers called **"Route Ciphers"** [7]. These were quite popular in the early history of cryptography. Generally, in Route Ciphers the elements of the plaintext (here single letters) are written on a pre-arranged route into a matrix agreed upon by the sender and the receiver. The above example uses a 2 (key = rows) by 19 (columns) matrix in which the plaintext is entered sequentially by columns, the encryption route is therefore to read the top row and then the lower.

In order to encrypt a message using the Transposition Cipher, the letters can be shuffled without any system. Otherwise, the recipient has no idea how to decrypt the

message. However, by choosing a system to rearrange the letters it allows an eavesdropper to be successful with his work. Testing many conceivable rearrangement will eventually the original message. In order to break the transposition cipher, the following steps can be followed:

Step 1: (Realization that the ciphertext was encrypted using a transposition cipher.) Computing the relative frequencies of the cipher letters reveals that cipher letters occur with the same frequency as plain letters. An eavesdropper realizes that plain letters were simply rearranged.

Step 2: Transposition ciphers are broken by testing possible rearrangements. First, try to read the cipher text backwards. If that does not yield the plain text then try the rail fence of depth two, then of depth three, then of depth four, etc. If that does not yield the plain text, check if two consecutive letters were switched. With today's computer power possible transpositions can be checked quickly. **In conclusion: the transposition ciphers don't offer any security**.

To gain an acceptable level of security, the route would have to be more complicated than the one in the above example. One form of transposition that has enjoyed widespread use relies on identifying the route by means of an easily remembered keyword. For example, by choosing the keyword "cat" a matrix can be written out like the one below. The letter "x" is used to fill the remaining spots in the matrix.

C t n o t n p r a e o n s y

A r s s i c h s n b k e i x

T a p i o i e c b r e a l x

The order in which the rows are read out to form the ciphertext is determined by the alphabetical order of the letters in the keyword "cat".

This matrix therefore yields the ciphertext:

"**rssichsnbkeixtnotnpraeonsyapioiecbrealx"**

To decode, the recipient simply fills 3 rows evenly with the ciphertext according to the alphabetical order of the letters in the shared keyword "cat".

The security of transposition ciphers can be further improved by re-encrypting the resulting cipher using another transposition. Because the product of the two transpositions is also a transposition, the effect of multiple transpositions is to further increase the complexity of the route through the matrix. However, although the plaintext gets more and more shuffled the plaintext letters are still part of the ciphertext and sufficient patience and experience in cryptoanalysis will eventually yield the original plaintext.

2.4 Symmetric-key Cryptosystem

2.4.1 Principles of Symmetric-key Cryptosystem

In **Symmetric-key cryptography**, a single key is used for both encryption and decryption. As shown in Figure 2.1, the sender uses the key (or some set of rules) to encrypt the plaintext and sends the ciphertext to the receiver. The receiver applies the same key (or ruleset) to decrypt the message and recover the plaintext. Because a single key is used for both functions, secret key cryptography is also called **symmetric encryption**.

With this form of cryptography, it is obvious that the key must be known to both the sender and the receiver; that, in fact, is the secret. The biggest difficulty with this approach, of course, is the distribution of the key.

Symmetric-key cryptography schemes are generally categorized as being either **stream ciphers** or **block ciphers** [7]. Stream ciphers operate on a single bit (byte or computer word) at a time, and implement some form of feedback mechanism so that the key is constantly changing. A block cipher is so-called because the scheme

encrypts one block of data at a time using the same key on each block. In general, the same plaintext block will always encrypt to the same ciphertext when using the same key in a block cipher whereas the same plaintext will encrypt to different ciphertext in a stream cipher.

Stream ciphers come in several flavors but two are widely used. **Self-synchronizing stream** *ciphers* calculate each bit in the key stream as a function of the previous *n* bits in the key stream. It is termed "self-synchronizing" because the decryption process can stay synchronized with the encryption process merely by knowing how far into the *n*-bit key stream it is. One problem is error propagation; a garbled bit in transmission will result in *n* garbled bits at the receiving side. **Synchronous stream ciphers** generate the key stream in a fashion independent of the message stream but by using the same key stream generation function at sender and receiver. While stream ciphers do not propagate transmission errors, they are, by their nature, periodic so that the key stream will eventually repeat.

Block ciphers can operate in one of several modes; the following four are the most important:

Electronic Codebook (ECB) *mode* is the simplest, most obvious application: the secret key is used to encrypt the plaintext block to form a ciphertext block. Two identical plaintext blocks, then, will always generate the same ciphertext block. Although this is the most common mode of block ciphers, it is susceptible to a variety of brute-force attacks.

Cipher Block Chaining (CBC) *mode* adds a feedback mechanism to the encryption scheme. In CBC, the plaintext is exclusively-ORed (XORed) with the previous ciphertext block prior to encryption. In this mode, two identical blocks of plaintext never encrypt to the same ciphertext.

Cipher Feedback (CFB) *mode* is a block cipher implementation as a self-synchronizing stream cipher. CFB mode allows data to be encrypted in units smaller than the block size, which might be useful in some applications such as encrypting

interactive terminal input. In case of 1-byte CFB mode, for example, each incoming character is placed into a shift register the same size as the block, encrypted, and the block transmitted. At the receiving side, the ciphertext is decrypted and the extra bits in the block (i.e., everything above and beyond the one byte) are discarded.

Output Feedback (OFB) *mode* is a block cipher implementation conceptually similar to a synchronous stream cipher. OFB prevents the same plaintext block from generating the same ciphertext block by using an internal feedback mechanism that is independent of both the plaintext and ciphertext bit streams.

2.4.2 Model of Symmetric-key Cryptosystem

A symmetric or conventional encryption scheme has five ingredients [8] (Figure 2.2)**:**

Plaintext /Message: This is the original intelligible message or data that is fed into the algorithm as input.

Encryption Algorithm: The encryption algorithm performs various substitution and transformation on the plaintext.

Secret Key: The secret key is also the input to the encryption algorithm. The key is a value independent of the plaintext. The algorithm will produce a different output depending on the specific key being used at the time. The exact substitutions and transformations performed by the algorithm depend on the key.

Ciphertext: This is the scrambled message produced as output. It depends on the plaintext and secret key. For a given message, two different keys will produce two different ciphertexts.

Decryption Algorithm: This is essentially the encryption algorithm run in reverse. It takes the ciphertext and the secret key and produces the original plaintext.

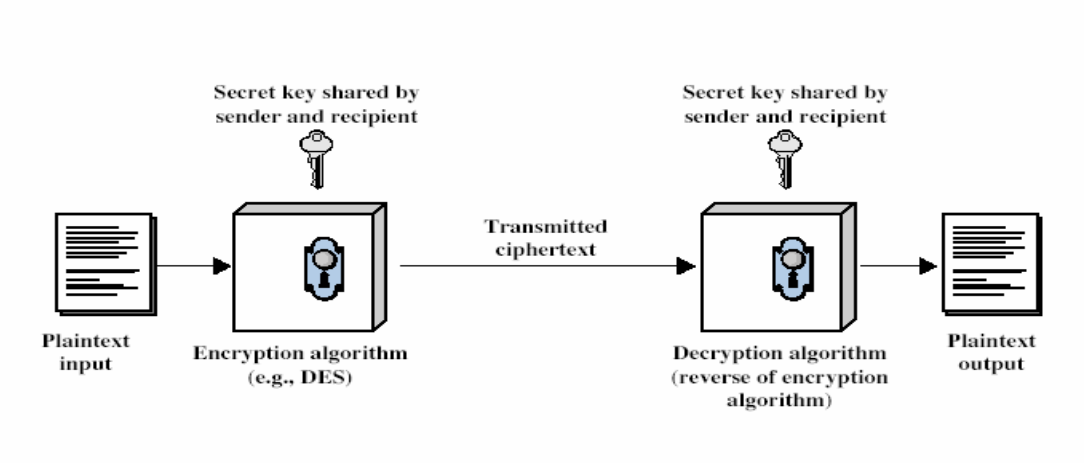


Figure 2.2 Model of Symmetric-key Cryptosystem

There are two requirements for secure use of symmetric-key encryption:

1. A strong encryption algorithm. At a minimum, the algorithm to be such that an opponent who knows the algorithm and has access to one or more ciphertexts would be unable to decipher the ciphertext or figure out the key. This requirement is usually stated in a stronger form: The opponent should be unable to decrypt ciphertext or discover the key even if he or she is in possession of a number of ciphertexts together with the plaintext that produced each ciphertext.

2. Sender and receiver must have obtained copies of the secret key in a secure fashion and must keep the key secure. If some can discover the key and knows the algorithm, all communication using this key is readable.

In symmetric-key cryptosystem, a source produces a message in plaintext, $\mathbf{X = [X_1, X_2, X_3, \ldots, X_M]}$. The M elements of X are letters in some finite alphabet. Traditionally, the alphabet usually consisted of the 26 capital letters. The binary alphabet **{0,1}** is also typically used. For encryption, a key of the form $\mathbf{K = [K_1, K_2, K_3, \ldots, K_J]}$ is generated. If the key is generated at the message source, then it must also be provided to the destination by means of some secure channel. Alternatively, a third party could generate the key and securely deliver it to both

source and destination. With the message X and the encryption key K as input, the encryption algorithm forms the ciphertext **Y = =[Y_1, Y_2, Y_3,. . . . ., Y_N].** This process can be expressed using the following notation

$$Y = E_k(X)$$

This notation indicates that Y is produced by using the encryption algorithm E as a function of the plaintext X, with the specific function determined by the value of the key. The intended receiver, in possession of the key, is able to invert the transformation:

$$X = E_k(Y)$$

2.4.3 Data Encryption Standard (DES)

The most common symmetric-key cryptography scheme used today is the **Data Encryption Standard *(DES)*,** designed by IBM in the 1970s and adopted by the National Bureau of Standards (NBS) [now the National Institute for Standards and Technology (NIST)] in 1977 [2] for commercial and unclassified government applications. DES has been adopted as Federal Information Processing Standard 46 (FIPS 46-3) and by the American National Standards Institute as X3.92. DES is a block-cipher employing a 56-bit key that operates on 64-bit blocks. DES has a complex set of rules and transformations that were designed specifically to yield fast hardware implementations and slow software implementations, although this latter point is becoming less significant today since the speed of computer processors is several orders of magnitude faster today than twenty years ago. IBM also proposed a 112-bit key for DES, which was rejected at the time by the government; the use of 112-bit keys was considered in the 1990s, however, conversion was never seriously considered.

Working Principle of DES:

DES uses a 56-bit key. In fact, the 56-bit key is divided into eight 7-bit blocks and an 8th odd parity bit is added to each block (i.e., a "0" or "1" is added to the block so

that there are an odd number of 1 bits in each 8-bit block). By using the 8 parity bits for rudimentary error detection, a DES key is actually 64 bits in length for computational purposes (although it only has 56 bits worth of randomness, or *entropy*).

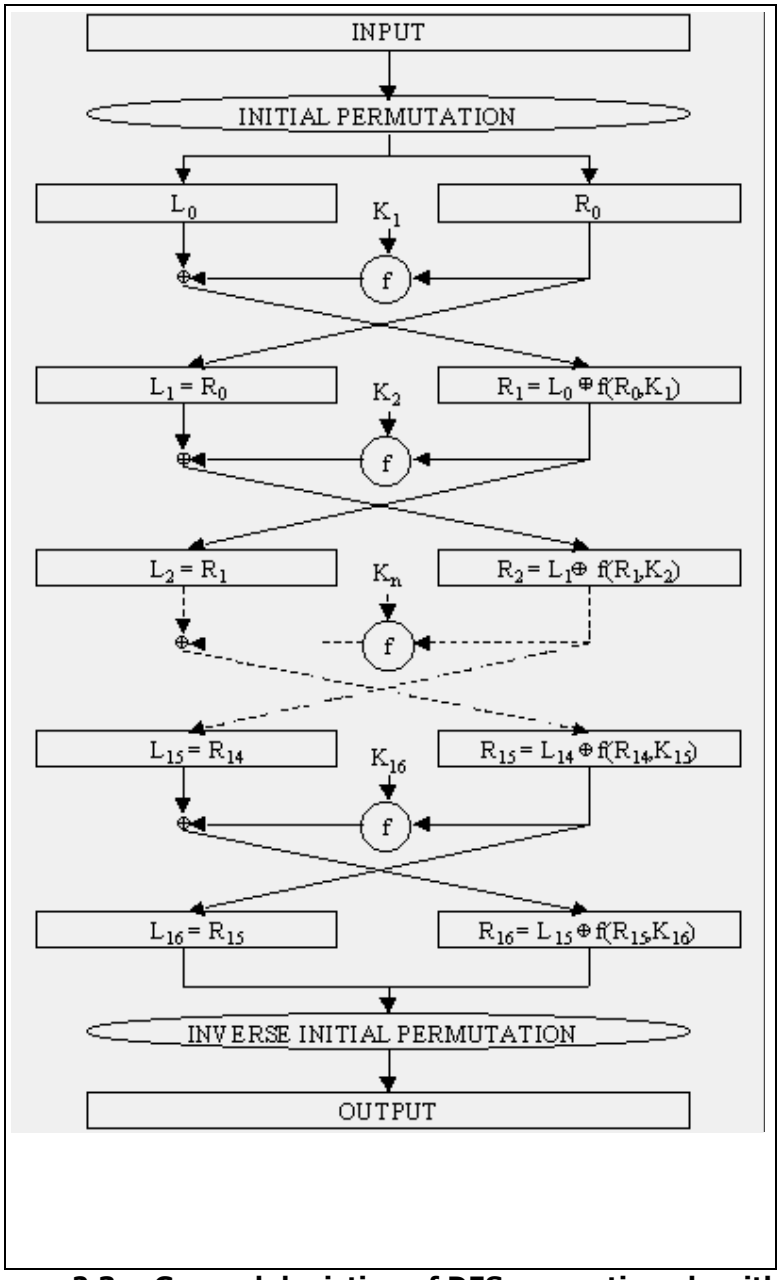


Figure 2.3 General depiction of DES encryption algorithm

DES then acts on 64-bit blocks of the plaintext, invoking 16 rounds of permutations, swaps, and substitutes, as shown in Figure 2.3. The standard includes tables

describing all of the selection, permutation, and expansion operations mentioned below; these aspects of the algorithm are not secrets. The basic DES steps are:

1. The 64-bit block to be encrypted undergoes an initial permutation (IP), where each bit is moved to a new bit position; e.g., the 1st, 2nd, and 3rd bits are moved to the 58th, 50th, and 42nd position, respectively.
2. The 64-bit permuted input is divided into two 32-bit blocks, called *left* and *right*, respectively. The initial values of the left and right blocks are denoted L_0 and R_0.
3. There are then 16 rounds of operation on the L and R blocks. During each iteration (where *n* ranges from 1 to 16), the following formulae apply:

   $L_n = R_{n-1}$

   $R_n = L_{n-1}$ XOR $f(R_{n-1},K_n)$

At any given step in the process, the new L block value is merely taken from the prior R block value. The new R block is calculated by taking the bit-by-bit exclusive-OR (XOR) of the prior L block with the results of applying the DES cipher function, *f*, to the prior R block and K_n. (K_n is a 48-bit value derived from the 64-bit DES key. Each round uses a different 48 bits according to the standard's Key Schedule algorithm.)

The cipher function, f, combines the 32-bit R block value and the 48-bit subkey in the following way. First, the 32 bits in the R block are expanded to 48 bits by an expansion function (E); the extra 16 bits are found by repeating the bits in 16 predefined positions. The 48-bit expanded R-block is then XORed with the 48-bit subkey. The result is a 48-bit value that is then divided into eight 6-bit blocks. These are fed as input into 8 selection (S) boxes, denoted $S_1,...,S_8$. Each 6-bit input yields a 4-bit output using a table lookup based on the 64 possible inputs; this results in a 32-bit output from the S-box. The 32 bits are then rearranged by a permutation function (P), producing the results from the cipher function.

4. The results from the final DES round — i.e., L_{16} and R_{16} — are recombined into a 64-bit value and fed into an inverse initial permutation (IP^{-1}). At this step, the bits are rearranged into their original positions, so that the 58th, 50th, and 42nd bits, for example, are moved back into the 1st, 2nd, and 3rd positions, respectively. The output from IP^{-1} is the 64-bit ciphertext block.

2.5 Public Key Cryptography

Symmetric-key cryptography is based on the sender and receiver of a message knowing and using the same secret key: the sender uses the secret key to encrypt the message, and the receiver uses the same secret key to decrypt the message. The main problem is getting the sender and receiver to agree on the secret key without anyone else finding out. If they are in separate physical locations, they must trust a courier, or a phone system, or some other transmission medium to prevent the disclosure of the secret key being communicated. Anyone who overhears or intercepts the key in transit can later read, modify, and forge all messages encrypted or authenticated using that key. The generation, transmission and storage of keys are called key management; all cryptosystems must deal with key management issues. Because all keys in a symmetric-key cryptosystem must remain secret, symmetric-key cryptography often has difficulty providing secure key management, especially in open systems with a large number of users.

The concept of **public-key cryptography** was introduced in 1976 by Whitfield Diffie and Martin Hellman [2] in order to solve the key management problem. In their concept, each person gets a pair of keys, one called the public key and the other called the *private key*. Each person's public key is published while the private key is kept secret. The need for the sender and receiver to share secret information is eliminated; all communications involve only public keys, and no private key is ever transmitted or shared. No longer is it necessary to trust some communications

channel to be secure against eavesdropping or betrayal. The only requirement is that public keys are associated with their users in a trusted (authenticated) manner (for instance, in a trusted directory). Anyone can send a confidential message by just using public information, but the message can only be decrypted with a private key, which is in the sole possession of the intended recipient. Furthermore, public-key cryptography can be used not only for privacy (*encryption*), but also for authentication (*digital signatures*).

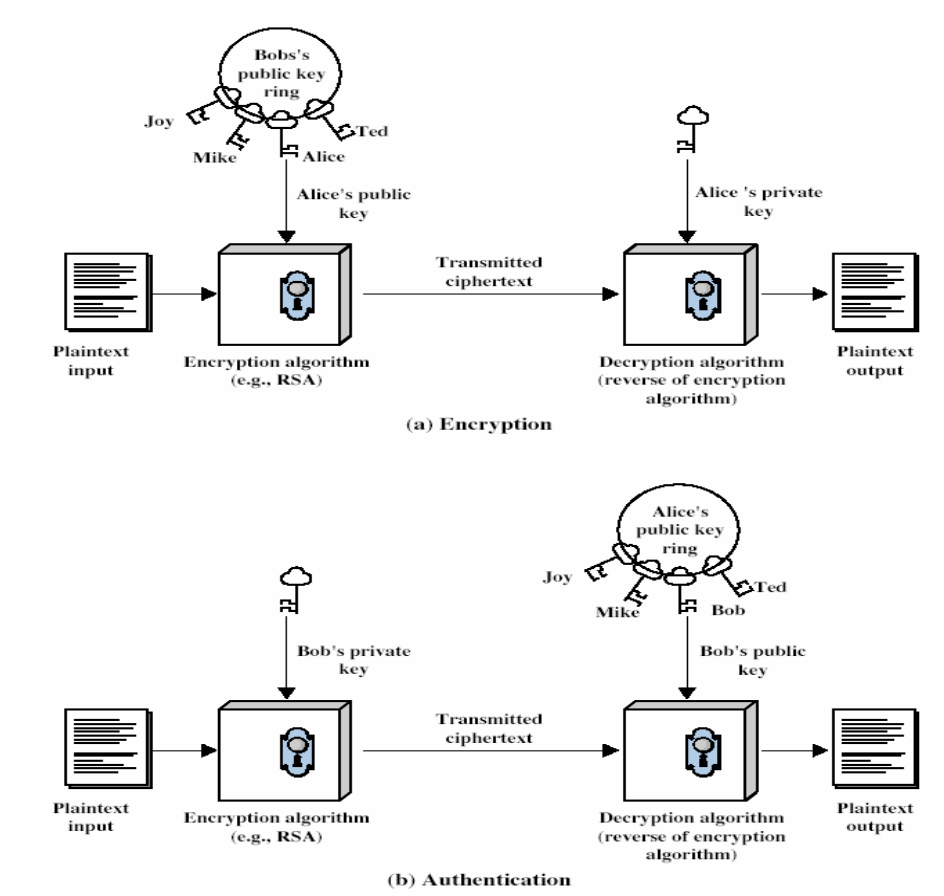


Figure 2.4 Public-key Cryptography

2.5.1 RSA Algorithm

The first, and still most common, Public-key Cryptosystem implementation is RSA, named for the three MIT mathematicians who developed it - Ron Rivest, Adi Shamir and Len Adleman, who invented it in 1977 [9]. The basic technique was first

discovered in 1973 by Clifford Cocks of CESG (part of the British GCHQ) but this was a secret until 1997. The RSA algorithm can be used for both public key encryption and digital signatures. Its security is based on the difficulty of factoring large integers.

The RSA scheme is a block cipher in which the plaintext and ciphertext are integers between 0 and n-1 for some n. A typical size of n is 1024 bits or 309 decimal digits. Plaintext is encrypted in blocks, with each block having a binary value less than some number n. That is, the block size must be less than or equal to $\log_2(n)$; in practice, the block size is k bits, where $2^k < n \leq 2^{k+1}$. Encryption and decryption are of the following form, for some plaintext block M and ciphertext block C:

$C = M^e \bmod n$

$M = C^d \bmod n = M^{ed} \bmod n$

Both sender and receiver must know the value of n. The sender knows the value of e, and only the receiver knows the value of d. Thus this is a public-key encryption algorithm with a public-key of KU ={e,n} and a private Key of KR ={d,n}. for This algorithm to be satisfactory for public-key encryption , the following requirements must be met:

- It is possible to find values of e,d,n such that $M^{ed} = M \bmod n$ for all $M < n$.
- It is relatively easy to calculate M^e and C^d for all values of $M < n$.
- It is infeasible to determine d given e and n.

Each participant must generate a pair of keys. This involves the following tasks:

- Selecting two prime numbers p and q.
- Calculating n = pq.
- Calculating $\Phi(n) = (p-1)(q-1)$.
- Selecting e such that e is relatively prime to $\Phi(n)$ and less than $\Phi(n)$.
- Determining d such that $de = 1 \bmod \Phi(n)$ and $d < \Phi(n)$.

2.6 Importance of Symmetric-key Cryptography

The primary advantage of public-key cryptography is increased security and convenience. Private keys never need to transmitted or revealed to anyone. In a symmetric-key system, by contrast, the symmetric keys must be transmitted (either manually or through a communication channel), and there may be a chance that an enemy can discover the symmetric keys during their transmission.

Another major advantage of public-key systems is that they can provide a method for **digital signatures**. Authentication via symmetric-key systems requires the sharing of some symmetric keys and sometimes requires trust of a third party as well. As a result, a sender can repudiate a previously authenticated message by claiming that the shared symmetric key was somehow compromised by one of the parties sharing the symmetric-key. Public-key authentication, on the other hand, prevents this type of repudiation; each user has sole responsibility for protecting his or her private key. This property of public-key authentication is often called **non-repudiation**.

A **disadvantage** of using public-key cryptography for encryption is **speed**; there are popular symmetric-key encryption methods that are significantly faster than any currently available public-key encryption method. Nevertheless, public-key cryptography can be used with symmetric-key cryptography to get the best of both worlds. **For encryption**, the best solution is to combine public- and symmetric-key systems in order to get both the security advantages of public-key systems and the speed advantages of symmetric-key systems. The public-key system can be used to encrypt a symmetric- key which is used to encrypt the bulk of a file or message. Such a protocol is called a *digital envelope*.

In some situations, public-key cryptography is not necessary and symmetric-key cryptography alone is sufficient. This includes environments where secure symmetric-

key agreement can take place, for example by users meeting in private. It also includes environments where a single authority knows and manages all the keys (e.g., a closed banking system). Since the authority knows everyone's keys already, there is not much advantage for some to be "public" and others "private." Also, public-key cryptography is usually not necessary in a single-user environment. In general, public-key cryptography is best suited for an open multi-user environment.

Public-key cryptography is not meant to replace symmetric-key cryptography, but rather to supplement it, to make it more secure. The first use of public-key techniques was for secure key exchange in an otherwise symmetric-key system; this is still one of its primary functions. Symmetric-key cryptography remains extremely important and is the subject of ongoing study and research.

Advantages of symmetric-key cryptography

1. Symmetric-key ciphers can be designed to have high rates of data throughput.
2. Keys for symmetric-key ciphers are relatively short.
3. Symmetric-key ciphers can be employed as primitives to construct various cryptographic mechanisms including pseudorandom number generators, hash functions, and computationally efficient digital signature schemes, to name just a few.
4. Symmetric-key ciphers can be composed to produce stronger ciphers. Simple transformations which are easy to analyze, but on their own weak, can be used to construct strong product ciphers.

Disadvantages of symmetric-key cryptography

1. In a two-party communication, the key must remain secret at both ends.
2. In a large network, there are many key pairs to be managed. Consequently, effective key management requires the use of an unconditionally trusted TTP.

3. In a two-party communication between entities A and B, sound cryptographic practice dictates that the key be changed frequently and perhaps for each communication session.

Digital signature mechanisms arising from symmetric-key encryption typically require either large keys for the public verification function or the use of a TTP.

Advantages of public-key cryptography

1. Only the private key must be kept secret (authenticity of public keys must, however, be guaranteed).
2. Depending on the mode of usage, a private key/public key pair may remain unchanged for considerable periods of time, e.g., many sessions (even several years).
3. Many public-key schemes yield relatively efficient digital signature mechanisms. The key used to describe the public verification function is typically much smaller than for the symmetric-key counterpart.
4. In a large network, the number of keys necessary may be considerably smaller than in the symmetric-key scenario.

Disadvantages of public-key encryption

1. Throughput rates for the most popular public-key encryption methods are several orders of magnitude slower than the best-known symmetric-key schemes.
2. Key sizes are typically much larger than those required for symmetric-key encryption, and the size of public-key signatures is larger than that of tags providing data origin authentication from symmetric-key techniques.

Summary of comparison

1. Public-key cryptography facilitates efficient signatures (particularly non-repudiation) and key management, and

2. Symmetric-key cryptography is efficient for encryption and some data integrity applications.

2.7 Limitations of DES

In cryptography, key size does matter. The larger the key, the harder it is to crack a block of encrypted data. The reason that large keys offer more protection is almost obvious; computers have made it easier to attack ciphertext by using brute force methods rather than by attacking the mathematics. With a brute force attack, the attacker merely generates every possible key and applies it to the ciphertext. Any resulting plaintext that makes sense offers a candidate for a legitimate key.

Until the mid-1990s or so, brute force attacks were beyond the capabilities of computers that were within the budget of the attacker community. Today, however, significant compute power is commonly available and accessible. General purpose computers such as PCs are already being used for brute force attacks. For serious attackers with money to spend, such as some large companies or governments, Field Programmable Gate Array (FPGA) or Application-Specific Integrated Circuits (ASIC) technology offers the ability to build specialized chips that can provide even faster and cheaper solutions than a PC. An AT&T ORCA chip (FPGA) costs $200 and can test 30 million DES keys per second, while a $10 ASIC chip can test 200 million DES keys per second (compared to a PC which might be able to test 40,000 keys per second).

The mainstream cryptographic community has long held that DES's 56-bit key was too short to withstand a brute-force attack from modern computers. DES is even more vulnerable to a brute-force attack because it is often used to encrypt words, meaning that the entropy of the 64-bit block is, effectively, greatly reduced. That is, if encryption is performed on random bit streams, then a given byte might contain any one of 2^8 (256) possible values and the entire 64-bit block has 2^{64}, or about 18.5 quintillion, possible values. If encryption is performed on words, however, it is most

likely to find a limited set of bit patterns; perhaps 70 or so if the upper and lower case letters, the numbers, space, and some punctuation are considered. This means that only about ¼ of the bit combinations of a given byte are likely to occur. In mid 1990, RSA Laboratories sponsored a series of cryptographic challenges to prove that DES was no longer appropriate for use .

DES Challenge I was launched in March 1997. It was completed in 84 days by R. Verser in a collaborative effort using thousands of computers on the Internet.

The first DES II challenge lasted 40 days in early 1998. This problem was solved by distributed.net, a worldwide distributed computing network using the spare CPU cycles of computers around the Internet (participants in distributed.net's activities load a client program that runs in the background. The distributed.net systems were checking 28 billion keys per second by the end of the project.

The second DES II challenge lasted less than 3 days. On July 17, 1998, the Electronic Frontier Foundation (EFF) announced the construction of hardware that could brute-force a DES key in an average of 4.5 days. Called Deep Crack, the device could check 90 billion keys per second and cost only about $220,000 including. Since the design is scalable, this suggests that an organization could build a DES cracker that could break 56-bit keys in an average of a day for as little as $1,000,000.

The DES III challenge, launched in January 1999, was broken is less than a day by the combined efforts of Deep Crack and distributed.net. This is widely considered to have been the final nail in DES's coffin.

2.8 AES: An Alternative to DES

From the above discussion, it is obvious that the symmetric-key cryptography is efficient for encryption while the Public-key cryptography facilitates efficient signatures (particularly non-repudiation) and key management. Symmetric–key

cryptography is faster than any currently available public-key encryption method. On the other hand, the most widely used symmetric-key encryption technique like DES is vulnerable to a brute-force attack because of its inadequate key size compare to the processing power of modern computer. In order to increase the security of symmetric-key cryptography, NIST in 1997 issued a call for proposals for a new Advanced Encryption Standard (AES), which should have security strength better than DES and significantly improved efficiency. In addition, to these general requirements, NIST specified that AES must be a symmetric block cipher with a block length of 128 bits and support for key lengths of 128, 192, and 256 bits. Chapter three presents working principle of AES.

CHAPTER 3

WORKING PRINCIPLE OF AES

The search for a replacement to DES started in January 1997 when NIST (National Institute of Standards and Technology) of the USA announced that it was looking for an **Advanced Encryption Standard**. In September of that year, they put out a formal Call for Algorithms and in August 1998 announced that 15 candidate algorithms were being considered (Round 1). In April 1999, NIST announced that the 15 had been whittled down to five finalists (Round 2): **MARS** (multiplication, addition, rotation and substitution) from IBM; Ronald Rivest's **RC6**; **Rijndael** from a Belgian team; **Serpent,** developed jointly by a team from England, Israel, and Norway; and **Twofish**, developed by Bruce Schneier. In October 2000, NIST announced their selection: **Rijndael**.

3.1 The AES Cipher

The Rijndael proposal for AES [11] defined a cipher in which the block length and the key length specified to be 128, 192, or 256 bits. The AES specification uses the same three key size alternatives but limits the block length to 128 bits. A number of AES parameters (Table 3.1) depend on the key length. Most of the implementation of AES uses the key length of 128 bits.

Table 3.1 AES parameter

Key size (words/byte/bits)	4/16/128	6/24/192	8/32/256
Plaintext block size (word/bytes/bits)	4/16/128	4/16/128	4/16/128
Number of rounds	10	12	14
Round key size (words/bytes/bits)	4/16/128	4/16/128	4/16/128
Expanded key size (words/bytes)	44/176	52/208	60/240

Rijndael was designed to have the following characteristics:

- Resistance against all known attacks.
- Speed and code compactness on a wide range of platforms.

- Design simplicity.

3.2 Overall Structure of AES

The overall structure of AES is depicted in figure 3.1. The input to the encryption and decryption algorithms is a single 128-bit block. This block of input is depicted as a square matrix of bytes. This block is copied into the **state** array, which is modified at each stage of encryption or decryption. After the final stage, **state** is copied to an output matrix. These operations are depicted in figure: 3.2. Similarly, the 128-bit key is depicted as a square matrix of bytes. This key is then expanded into an array of key schedule words; each word is four bytes and total key schedule is 44 words for the 128-bit key. The ordering of bytes within a matrix is by column. So, for example, the first four bytes of a 128-bit plaintext input to the encryption cipher occupy the first column of the **in** matrix, the second four bytes occupy the second column, and so on. Similarly, the first four bytes of the expanded key, which form a word, occupy the first column of the **w** matrix.

Several features of the overall AES structure [8]:

1. One noteworthy feature of this structure is that it is not a Feistel structure. In the classic feistel structure, half of the data block is used to modify the other half of the data block, and then the half are swapped. Rijndael does not use a Feistel structure but process the entire block in parallel during each round using substitutions and permutations.
2. The key that is provided as input is expanded into an array of forty-four 32-bit words, w[i]. Four distinct words (128 bits) serve as a round key for each round; these are indicated in Figure 3.1.

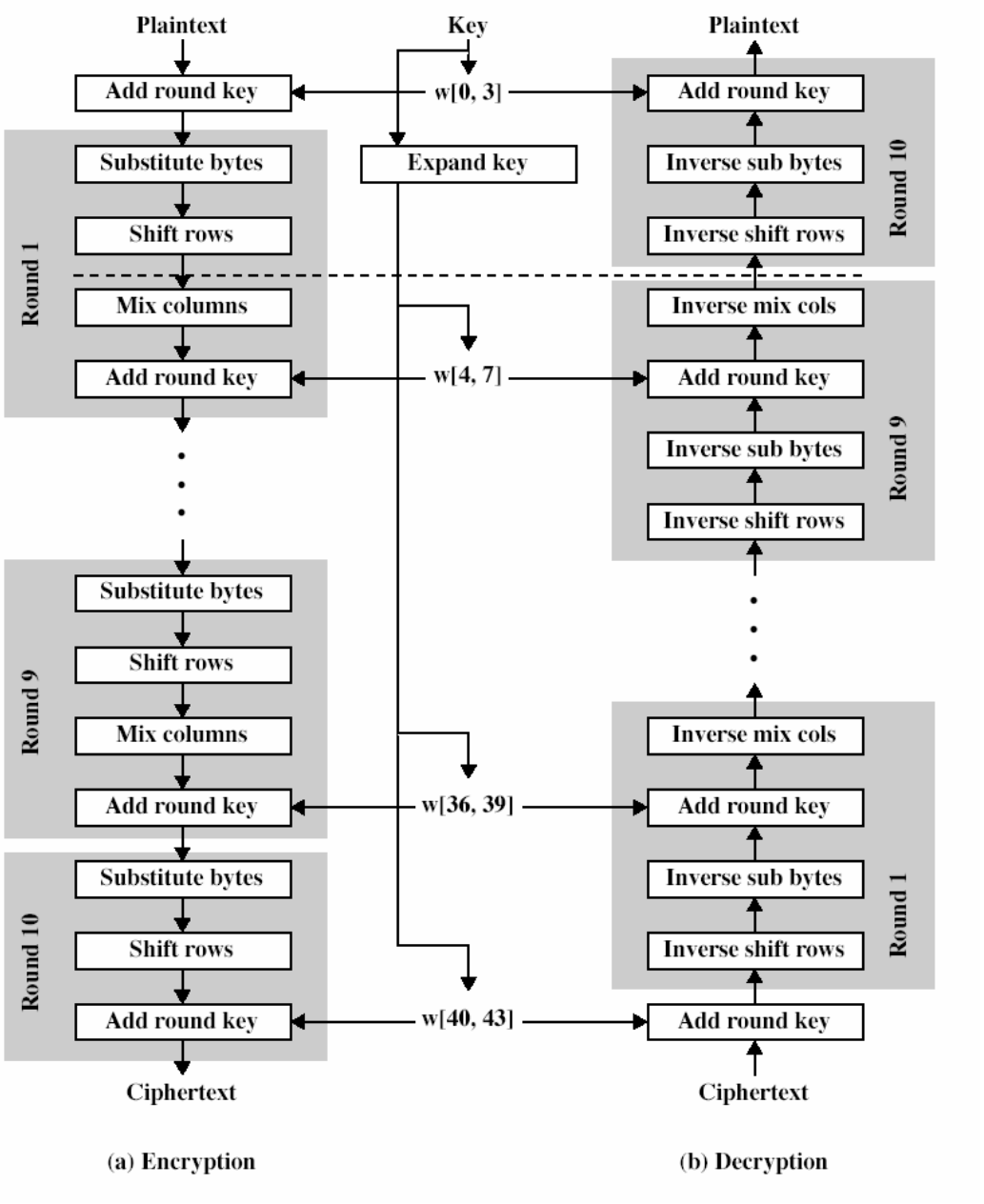


Figure 3.1 AES encryption and decryption

3. Four different stages are used, one of permutation and three of substitution:

   Substitute bytes: Uses an S-box to perform a byte-by-byte substitution of the block

   Shift rows: A simple permutation

   Mix Columns: A substitution that makes use of arithmetic over GF (2^8)

Add round Key: A simple bitwise XOR of the current block with a portion of the expanded key

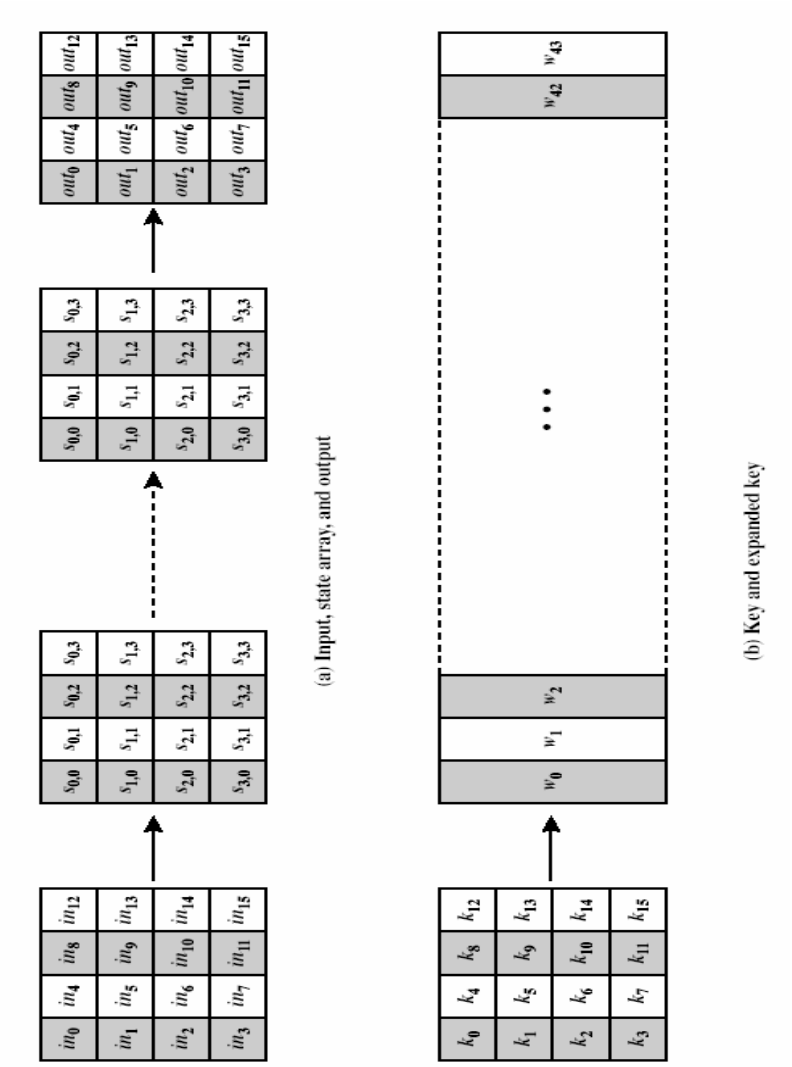


Figure 3.2 AES data structures

4. The structure of AES is quite simple. For both encryption and decryption, the cipher begins with an **Add Round Key stage**, followed by nine rounds that each includes all four stages, followed by a tenth round of three stages. Figure 3.3 depicts the structure of a full encryption round.
5. Only the Add Round Key stage makes use of the key. For this reason, the cipher begins and ends with an Add Round Key stage. Any other stage, applied at the beginning or end, is reversible without knowledge of the key

and so would add no security.

6. The Add Round Key stage is, in effect, a form of Vernam cipher and by itself would not be formidable. The other three stages together provide confusion, diffusion, and nonlinearity, but by themselves would provide no security because they do not use the key. The cipher is an alternating operations of XOR encryption (Add Round Key) of a block, followed by scrambling of the block (the other three stages), and followed by XOR encryption, and so on. This scheme is both efficient and highly secure.
7. Each stage is easily reversible. For the Substitute Byte, Shift Row, and Mix Columns stages, an inverse function is used in the decryption algorithm. For the Add Round Key stage, the inverse is achieved by XORing the same round key to the block, using the result that $A \oplus A \oplus B = B$.
8. As with most block ciphers, the decryption algorithm makes use of the expanded key in reverse order. However, the decryption algorithm is not identical to the encryption algorithm. This is a consequence of the particular structure of AES.
9. Once it is established that all four stages are reversible, it is easy to verify that decryption does recover the plaintext. Figure 3.1 lays out encryption and decryption going in opposite vertical directions. At each horizontal point (e.g., the dashed line in the figure), **State** is the same for both encryption and decryption.
10. The final round of both encryption and decryption consists of only three stages. Again, this is a consequence of the particular structure of AES and is required to make the cipher reversible.

Now, all the four stages that is used in AES will be discussed. For each stage, the forward (encryption) algorithm, the inverse (decryption) algorithm, and the rational for the stage will be described. This is followed by a discussion of key expansion.

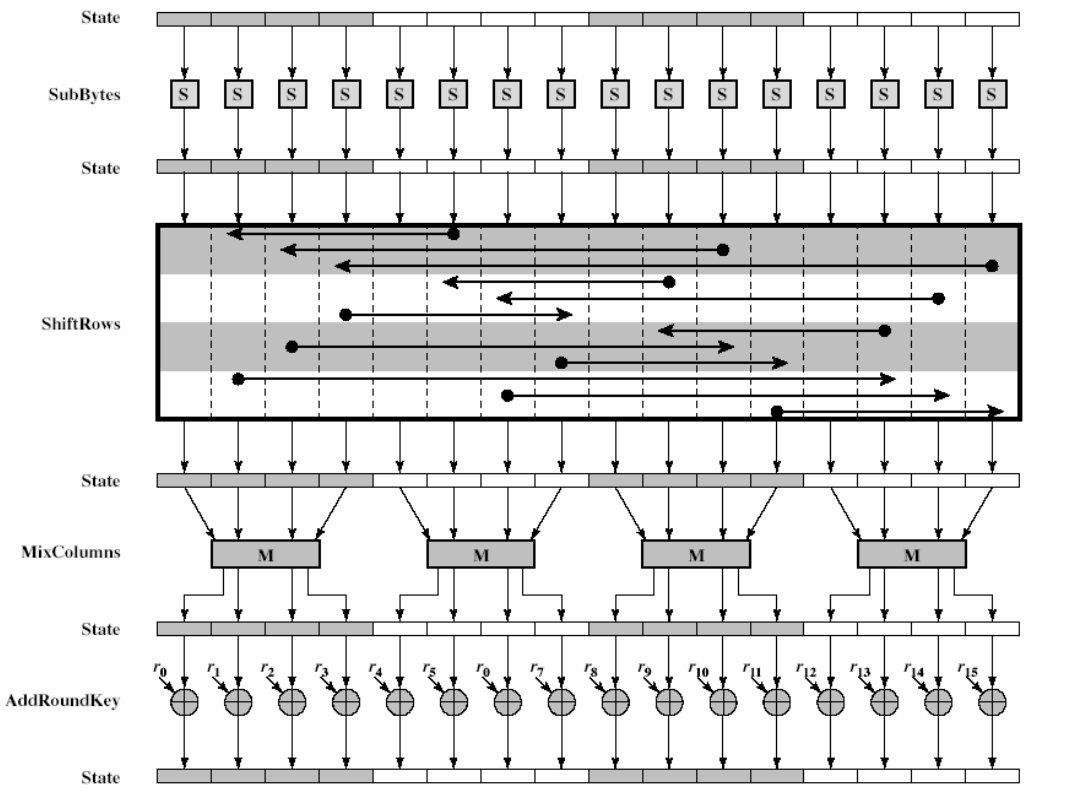
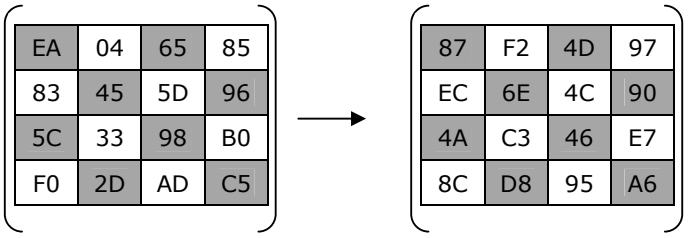


Figure 3.3 AES encryption round

3.3 Substitute Bytes Transformation

Forward and Inverse Transformations

The **forward substitute byte transformation,** called SubBytes [12], is a simple table lookup (Figure 3.4a). AES defines a 16 X 16 matrix of byte values, called an S-box (Table 3.2a), that contains a permutation of all possible 256 8-bit values. Each individual byte of **State** is mapped into a new byte in the following way: The leftmost 4 bits of the byte are used as a row value and the rightmost 4 bits are used as a column value. These row and column values serve as indexes into the S-box to select a unique 8-bit output value. For example, the hexadecimal value {95} references row 9, column 5 of the S-box, which contains the value {2A}. Accordingly, the value {95} is mapped into the value {2A}. Here is an example of the SubBytes transformation:



The S-box is constructed in the following fashion:

1. Initialize the S-box with the byte values in ascending sequence row by row. The first row contains {00}, {01}, {02}, . . . , {0F}; the second row contains {10}, {11}, etc.; and so on. Thus, the value of the byte at row *x,* column *y* is *{xy}.*
2. Map each byte in the S-box to its multiplicative inverse in the finite field GF(2^8); the value {00} is mapped to itself.
3. Each byte in the S-box consists of 8 bits labeled *(b7, b6, b5, b4, b3, h2, b1, bo).* The following transformation is applied to each bit of each byte in the S-box:

$$b_i' = b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus C_i \qquad \textbf{3.1}$$

where C_i is the ith bit of byte C with the value {63}; that is, $(C_7C_6C_5C_4C_3C_2C_1C_0)$ = (01100011). The prime (') indicates that the variable is to be updated by the value on the right. The AES standard depicts this transformation in matrix form as follows:

$$\begin{pmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

3.2

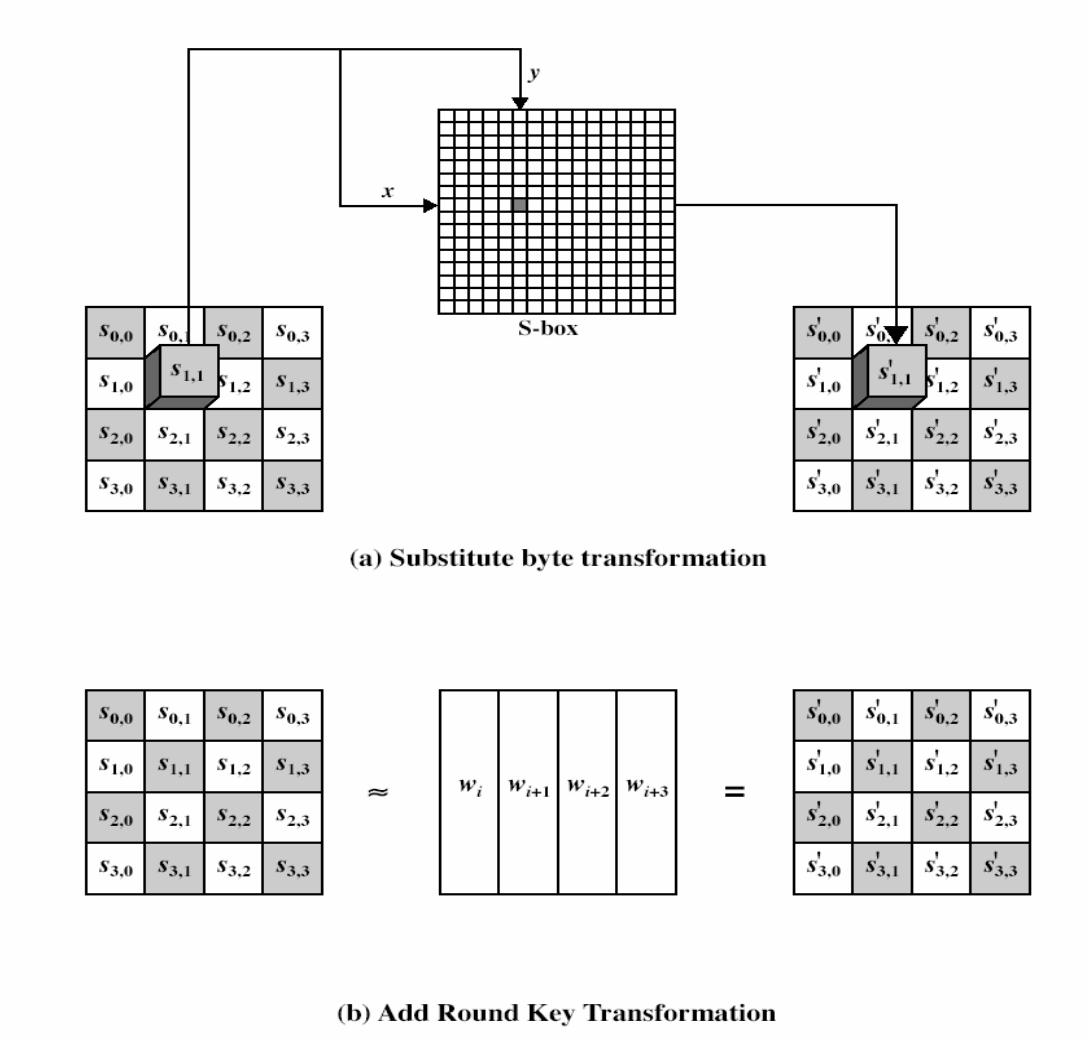


Figure 3.4 AES byte-level operations

In ordinary matrix multiplication, each element in the product matrix is the sum of products of the elements or one row and one column. In this case, each element in the product matrix is the bitwise XOR of products of elements of one row and one column. Further, the final addition shown in the Equation 3.2 is a bitwise XOR.

Table 3.2 AES S-boxes

(a) S-box

		Y															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
X	**0**	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
	1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
	2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
	3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
	4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
	5	53	Dl	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
	6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
	7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
	8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
	9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
	A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
	B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
	C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
	D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
	E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
	F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

(b) Inverse S-box

		y															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
X	**0**	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
	1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
	2	54	7B	94	32	A6	C2	23	3D	EE	*AC*	95	0B	42	FA	C3	4E
	3	08	2E	Al	66	28	D9	24	B2	76	5B	A2	49	6D	8B	Dl	25
	4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
	5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
	6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
	7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
	8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
	9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
	A	47	Fl	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
	B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
	C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
	D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
	E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
	F	17	2B	04	7E	BA	77	D6	26	El	69	14	63	55	21	0C	7D

As an example, the input value {95} is considered. The multiplicative inverse in GF(2^8) is $\{95\}^{-1}$ = (8A), which is 10001010 in binary. Using the above Equation,

$$\begin{pmatrix} 1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1 \end{pmatrix} \begin{pmatrix} b_0\\ b_1\\ b_2\\ b_3\\ b_4\\ b_5\\ b_6\\ b_7 \end{pmatrix} \oplus \begin{pmatrix} b_0\\ b_1\\ b_2\\ b_3\\ b_4\\ b_5\\ b_6\\ b_7 \end{pmatrix} = \begin{pmatrix} b_0\\ b_1\\ b_2\\ b_3\\ b_4\\ b_5\\ b_6\\ b_7 \end{pmatrix} \oplus \begin{pmatrix} b_0\\ b_1\\ b_2\\ b_3\\ b_4\\ b_5\\ b_6\\ b_7 \end{pmatrix} = \begin{pmatrix} b_0\\ b_1\\ b_2\\ b_3\\ b_4\\ b_5\\ b_6\\ b_7 \end{pmatrix}$$

The result is (2A), which will appear in row {09} column {05} of the S-box. This is verified by checking Table 3.2a.

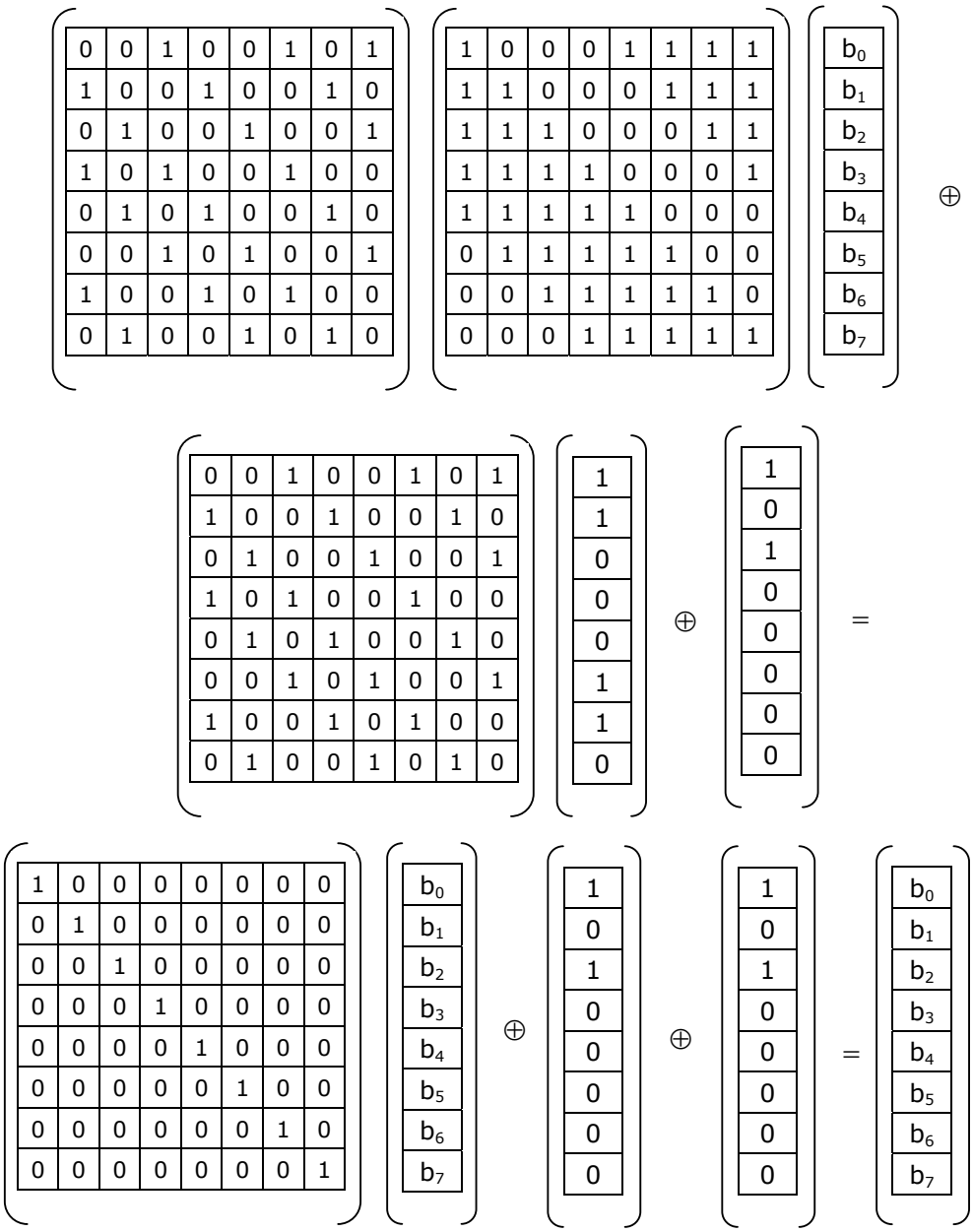
The **inverse substitute byte transformation,** called **InvSubBytes**, makes use of the inverse S-box shown in Table 3.2b. The input {2AJ produces the output {95J, and the input (95J to the S-box produces {2A}. The inverse S-box is constructed by applying the inverse of the transformation in Equation (3.1) followed by taking the multiplicative inverse in GF(2^8). The inverse transformation is

$$b_i' = b_i \oplus b_{(i+2) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus \mathbf{d}_i$$

where byte d= {05}, or 00000101. It can be represented as follows:

$$\begin{pmatrix} b'_0\\ b'_1\\ b'_2\\ b'_3\\ b'_4\\ b'_5\\ b'_6\\ b'_7 \end{pmatrix} = \begin{pmatrix} 0&0&1&0&0&1&0&1\\ 1&0&0&1&0&0&1&0\\ 0&1&0&0&1&0&0&1\\ 1&0&1&0&0&1&0&0\\ 0&1&0&1&0&0&1&0\\ 0&0&1&0&1&0&0&1\\ 1&0&0&1&0&1&0&0\\ 0&1&0&0&1&0&1&0 \end{pmatrix} \begin{pmatrix} b_0\\ b_1\\ b_2\\ b_3\\ b_4\\ b_5\\ b_6\\ b_7 \end{pmatrix} + \begin{pmatrix} 1\\ 0\\ 1\\ 0\\ 0\\ 0\\ 0\\ 0 \end{pmatrix}$$

To verify that **InvSubBytes** is the inverse of **SubBytes**, the matrices in **SubBytes** and **InvSubBytes** are labeled as **X** and **Y,** respectively, and the vector versions of constants c and d are labeled as **C** and **D,** respectively. For some 8-bit vector **B,** Equation (3.2) becomes B' = **XB** ⊕ C. It must be proved that **Y(XB** ⊕ C) ⊕ D = **B.** Multiply out, It must satisfy that **YXB** ⊕ **YC** ⊕ **D** = B. This becomes



It is proved from the above equation that **YX** equals to the identity matrix, and the **YC** = **D**, so that **YC** ⊕ **D** equals the null vector.

The S-box is designed to be resistant to known cryptanalytic attacks. Specifically, the Rijndael developers sought a design that has a low correlation between input bits and outputs bits, and the property that the output cannot be described as a simple mathematical function of the input. In addition, the constant in Equation (3.1) was chosen so that the S-box has no fixed points [S-box(a)=a] and no "opposite fixed points" [S=box(a)=ā], where ā is the bitwise complement of a.

The S-box must be invertible, that is, IS-box[S-box(a)]=a. However, the S-box is not self-inverse in the sense that it is not true that S-box(a)=IS-box(a). For example, S-box({95})={2A}, but IS-box ({95})={AD}.

3.4 Shift Row Transformation

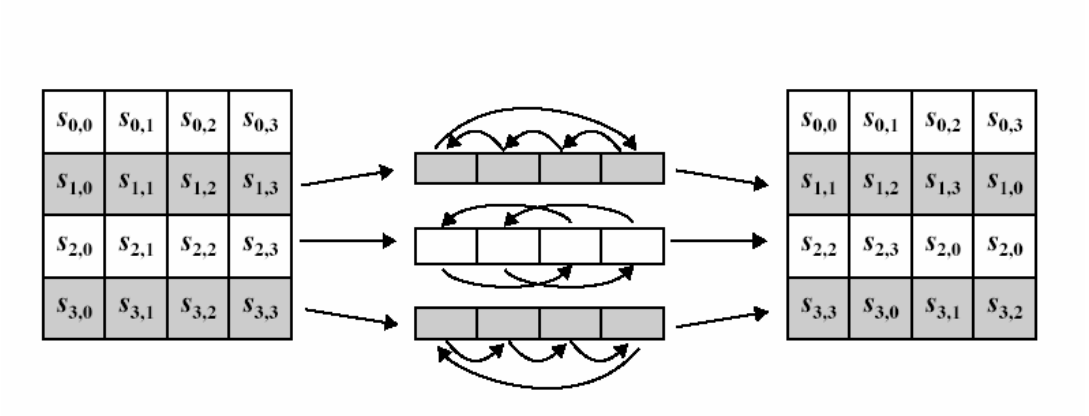


Figure 3.5 Shift row transformation

The **forward shift row transformation**, called **ShiftRows** [11], is depicted in Figure 3.5. The first row of state is not altered. For the second row, a 1-byte circular left shift is performed. For the third row, a 2-byte circular left shift is performed. For the fourth row, a 3-byte circular left shift is performed. The following is an example of **shiftRows**:

87	F2	4D	97
6E	4C	90	EC
46	E7	4A	C3
A6	8C	D8	95

→

87	F2	4D	97
EC	6E	4C	90
4A	C3	46	E7
8C	D8	95	A6

The **inverse shift row transformation,** called **InvShiftRows**, performs the circular shifts in the opposite direction for each of the last three rows, with a one-byte circular right shift for the second row, and so on.

The shift row transformation is more substantial than it may first appear. This is because the **State,** as well as the cipher input and output, is treated as an array of four 4-byte columns. Thus, on encryption, the first 4 bytes of the plaintext are copied

to the first column of **State,** and so on. However, the round key is applied to **State** column by column. Thus, a row shift moves an individual byte from one column to another, which is a linear distance of a multiple of 4 bytes. Moreover, the transformation ensures that the 4 bytes of one column are spread out to four different columns.

3.5 Mix Column Transformation

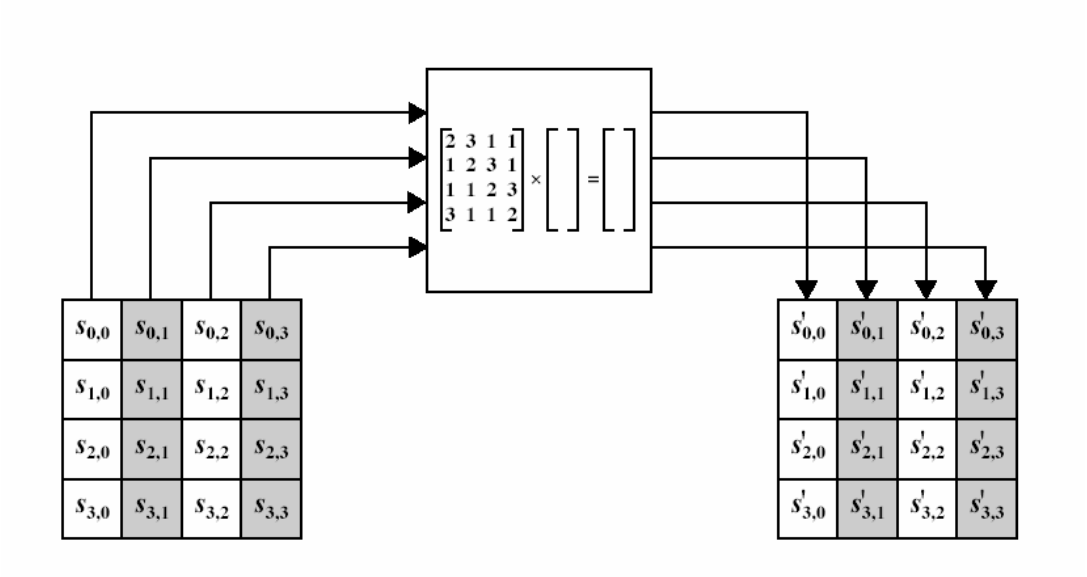


Figure 3.6 Mix column transformation

The **forward mix column transformation,** called MixColumns [12], operates on each column individually. Each byte of a column is mapped into a new value that is a function of all four bytes in the column. The transformation can be defined by the following matrix multiplication on **State.**

02	03	01	01
01	02	03	01
01	01	02	03
03	01	01	02

$S_{0,0}$	$S_{0,1}$	$S_{0,2}$	$S_{0,3}$
$S_{1,0}$	$S_{1,1}$	$S_{1,2}$	$S_{1,3}$
$S_{2,0}$	$S_{2,1}$	$S_{2,2}$	$S_{2,3}$
$S_{3,0}$	$S_{3,1}$	$S_{3,2}$	$S_{3,3}$

=

$S'_{0,0}$	$S'_{0,1}$	$S'_{0,2}$	$S'_{0,3}$
$S'_{1,0}$	$S'_{1,1}$	$S'_{1,2}$	$S'_{1,3}$
$S'_{2,0}$	$S'_{2,1}$	$S'_{2,2}$	$S'_{2,3}$
$S'_{3,0}$	$S'_{3,1}$	$S'_{3,2}$	$S'_{3,3}$

(3.3)

Each element in the product matrix is the sum of products of elements of one row and one column. In this case, the individual additions and multiplications are performed in GF(2^8). The MixColumns transformation on a single column j ($0 \leq j \leq 3$) **of**

State can be expressed as

$$S'_{0,j} = (2 \bullet S_{0,j}) \oplus (3 \bullet S_{1,j}) \oplus S_{2,j} \oplus S_{3,j}$$

$$S'_{1,j} = S_{0,j} \oplus (2 \bullet S_{1,j}) \oplus (3 \bullet S_{2,j}) \oplus S_{3,j}$$

$$S'_{2,j} = S_{0,j} \oplus S_{1,j} \oplus (2 \bullet S_{2,j}) \oplus (3 \bullet S_{3,j})$$

$$S'_{2,j} = (3 \bullet S_{0,j}) \oplus S_{1,j} \oplus S_{2,j} \oplus (2 \bullet S_{3,j})$$

The following is an example of MixColumns:

87	F2	4D	97
6E	4C	90	EC
46	E7	4A	C3
A6	8C	D8	95

→

47	40	A3	4C
37	D4	70	9F
94	E4	3A	42
ED	A5	A6	BC

The first column of the above example will be verified now. In GF(2^8), addition can be implemented by bitwise XOR operation and multiplication by a value (i.e., by {02}) can be implemented as a 1-bit left shift followed by a conditional bitwise XOR with (00011011) if the leftmost bit of the original value (prior to the shift) is 1. Thus, to verify the MixColumns transformation on the first column, the following equations must be verified:

$$\{02\} \bullet \{87\}) \oplus (\{03\} \bullet \{6E\}) \oplus \{46\} \oplus \{A6\} = \{47\}$$

$$\{87\}) \oplus (\{02\} \bullet \{6E\}) \oplus (\{03\} \bullet \{46\}) \oplus \{A6\} = \{37\}$$

$$\{87\}) \oplus \{6E\} \oplus (\{02\} \bullet \{46\}) \oplus (\{02\} \bullet \{A6\}) = \{94\}$$

$$(\{03\} \bullet \{87\}) \oplus \{6E\} \oplus \{46\} \oplus (\{02\} \bullet \{A6\}) = \{ED\}$$

For the first equation, {02} • {87} = (0000 1110) ⊕ (0001 1011) = (0001 0101); and {03} • {6E}= {6E} ⊕ ({02) • {6E }) = (0110 1110) ⊕ (1101 1100) = (1011 0010). Then

{02} • {87}	=	0001 0101
{03} • {6E}	=	1011 0010
{46}	=	0100 0110
{A6}	=	1010 0110
		0100 0111

The other equations can be similarly verified.

The **inverse mix column transformation,** called InvMixColumns, is defined by the following matrix multiplication:

$$\begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \begin{pmatrix} S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\ S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} \\ S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} \\ S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} \end{pmatrix} = \begin{pmatrix} S'_{0,0} & S'_{0,1} & S'_{0,2} & S'_{0,3} \\ S'_{1,0} & S'_{1,1} & S'_{1,2} & S'_{1,3} \\ S'_{2,0} & S'_{2,1} & S'_{2,2} & S'_{2,3} \\ S'_{3,0} & S'_{3,1} & S'_{3,2} & S'_{3,3} \end{pmatrix} \quad \textbf{(3.5)}$$

It is not immediately clear that Equation (3.5) is the inverse of equation (3.3). To show that

$$\begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\ S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} \\ S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} \\ S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} \end{pmatrix} =$$

$$\begin{pmatrix} S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\ S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} \\ S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} \\ S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} \end{pmatrix}$$

which is equivalent to showing that

$$\begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \textbf{(3.6)}$$

That is, the inverse transformation matrix times the forward transformation matrix equals the identity matrix. To verify the first column of Equation (3.6), the following equations must be verified

$$\textbf{(\{0E\} • \{02\})} \oplus \textbf{(\{0B\} • \{0D\})} \oplus \textbf{\{09\}} \oplus \textbf{\{03\}} = \textbf{\{01\}}$$

$$\textbf{(\{09\} • \{02\})} \oplus \textbf{(\{0E\} • \{0B\})} \oplus \textbf{\{0D\}} \oplus \textbf{\{03\}} = \textbf{\{00\}}$$

$$\textbf{(\{0D\} • \{02\})} \oplus \textbf{(\{09\} • \{0E\})} \oplus \textbf{\{0B\}} \oplus \textbf{\{03\}} = \textbf{\{00\}}$$

$$(\{0B\} \bullet \{02\}) \oplus (\{0D\} \bullet \{09\}) \oplus \{0E\} \oplus \{03\} = \{00\}$$

For the first equation, {0E} •{02}= 0001 1100; and {09} • {03} = {09} ⊕ ({09} • {02}) = 00001001 ⊕ 00010010 = 00011011. Then

{0E} • {02}	=	0001 1100
{0B}	=	0000 1011
{0D}	=	0000 1101
{09} • {03}	=	0001 1011
		0000 0001

The other equations can be similarly verified.

The coefficients of the matrix in Equation (3.3) are based on a linear code with maximal distance between code words, which ensures a good mixing among the bytes of each column. The mix column transformation combined with the shift row transformation ensures that after a few rounds, all output bits depend on all input bits [DAEM99].

In addition, the choice of coefficients in **MixColumns**, which are all {01}, {02}, or {03}, was influenced by implementation considerations. As was discussed, multiplication by these coefficients involves at most a shift and an XOR. The coefficients in **InvMixColumns** are more formidable to implement. However, encryption was deemed more important than encryption for two reasons:

1. For the CFB and OFB cipher modes , only encryption is used.
2. As with any block cipher, AES can be used to construct a message authentication code, and for this only encryption is used.

3.6 Add Round Key Transformation

In the **forward add round key transformation,** called **AddRoundKey**, the 128 bits of **State** are bitwise XORed with the 128 bits of the round key. As shown in Fig-

ure 3.4(b), the operation is viewed as a colunmwise operation between the 4 bytes of **a State** column and one word of the round key; it can also be viewed as a byte-level operation. The following is an example of AddRoundKey:

47	40	A3	4C	⊕	AC	19	28	57	=	EB	59	8B	1B
37	D4	70	9F		77	FA	D1	5C		40	2E	A1	C3
94	E4	3A	42		66	DC	29	00		F2	38	13	42
ED	A5	A6	BC		F3	21	41	6A		1E	84	E7	D2

The first matrix is State, and the second matrix is the round key.

The inverse add round key transformation is identical to the forward add round key transformation, because the XOR operation is its own inverse.

The add round key transformation is as simple as possible and affects every bit of **State**. The complexity of the round key expansion, plus the complexity of the other stages of AES ensure security.

3.7 AES Key Expansion

Key Expansion Algorithm

The AES key expansion algorithm [11] takes as input a 4-word (16-byte) key and produces a linear array of 44 words (156 bytes). This is sufficient to provide a 4-word round key for the initial Add Round Key stage and each of the 10 rounds of the cipher. The following pseudocode describes the expansion:

```
KeyExpansion (byte key[16], word w[44]){
        word temp
        for (i = 0; i < 4; i + +)
                w[i] = (key[4*i],key[4*i + 1], key[4*i + 2], key[4*i + 3]);
        for (i = 4; i < 44; i+ + ){
                temp = w[i - 1];
                if (i mod 4 = 0)     temp = SubWord (RotWord (temp)) ^ Rcon[i/4];
                w[i] = w[i-4] ^ temp
        }
}
```

The key is copied into the first four words of the expanded key. The remainder of the expanded key is filled in four words at a time. Each added word w[i] depends on the immediately preceding word, w[i - 1], and the word four positions back, w[i - 4]. In three out of four cases, a simple XOR is used. For a word whose position in the w array is a multiple of 4, a more complex function is used. Figure 3.7 illustrates the generation of the first eight words of the expanded key, using the symbol g to represent that complex function. The function g consists of the following **subfunctions**:

1. **RotWord** performs a one-byte circular left shift on a word. This means that an input word b0, b1, b2, b3] is transformed into [b1, b2, b3, bO].
2. **SubWord** performs a byte substitution on each byte of its input word, using the S-box (Table 3.2a).
3. The result of steps 1 and 2 is XORcd with a round constant, Rcon[j].

The round constant is a word in which the three rightmost bytes are always O. Thus the effect of an XOR of a word with Rcon is to only perform an XOR on the leftmost byte of the word.

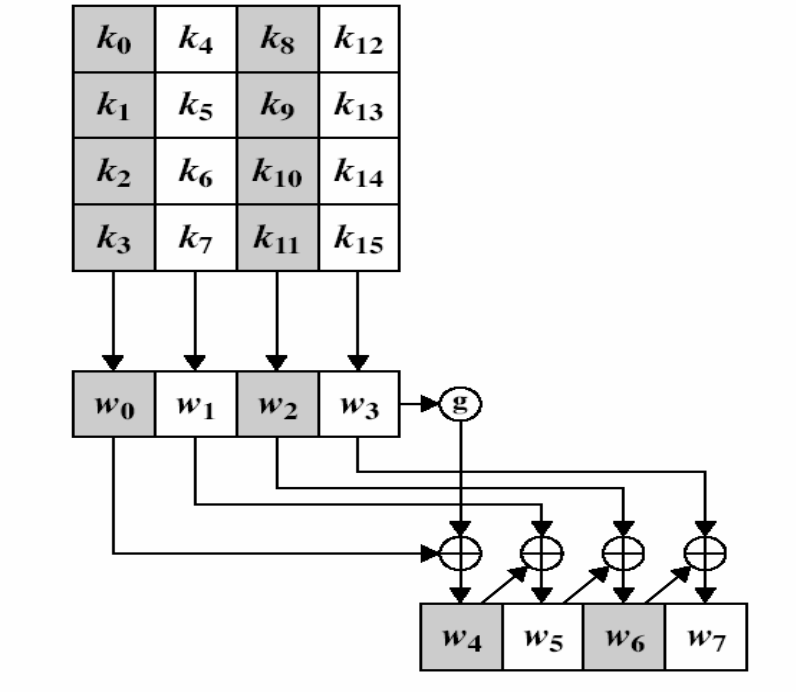


Figure 3.7 AES key expansion

The round constant is different for each round and is defined as Rcon[j] = (RC[j], 0, 0, 0), with RC[1] = 1, RC[j] = 2 • RC[j - 1] and with multiplication defined over the field GF(2^8). The values of RC[j] in hexadecimal are

J	1	2	3	4	5	6	7	8	9	10
RC[jJ	01	02	04	08	10	20	40	80	1B	36

For example, suppose that the round key for round 8 is

EA D2 73 21 B5 8D BA D2 31 2B F5 60 7F 8D 29 2F

Then the first 4 bytes (first column) of the round key for round 9 are calculated as follows:

i decimal	temp	After RotWord	After SubWord	Rcon (9)	After XOR with Rcon	w[i - 4J	w[iJ = temp ⊕ w[i - 4]
36	7F8D292F	8D292F7F	SDAS1SD2	1B000000	46ASlSD2	EAD27321	AC7766F3

The Rijndael developers designed the expansion key algorithm to be resistant to known cryptanalytic attacks. The inclusion of a round-dependent round constant eliminates the symmetry, or similarity between the way in which round keys are generated in different rounds. The specific criteria that were used are [13] as follows:

- Knowledge of a part of the cipher key or round key does not enable calculation of many other round key bits
- An invertible transformation [i.e., knowledge of any *Nk* consecutive words of the Expanded Key enables regeneration the entire expanded key (*Nk*=Key size in words)]
- Speed on a wide range of processors
- Usage of round constants to eliminate symmetries
- Diffusion of cipher key differences into the round keys; that is, each key bit affects many round key bits
- Enough nonlinearity to prohibit the full determination of round key differences from cipher key differences only
- Simplicity of description

If one knows less than Nk consecutive words of either the cipher key or one of the round keys, then it is difficult to reconstruct the remaining unknown bits. The fewer bits one knows, the more difficult it is to do the reconstruction or to determine other bits in the key expansion.

3.8 Equivalent Inverse Cipher

The AES decryption cipher is not identical to the encryption cipher (Figure 3.1), That is, the sequence of transformation for decryption differs from that for encryption, although the form of the key scheduled for encryption and decryption is the same. This has the disadvantage that two separate software or firmware modules are needed for applications that require both encryption and decryption. There is, however, an equivalent version of the decryption algorithm that has the same structure as the encryption algorithm. The equivalent version has the same sequence of transformation as the encryption algorithm (with transformations replaced by their inverses). To achieve this equivalence, a change in key schedule is needed.

Two separate changes are needed to bring the decryption structure in line with the encryption structure. An encryption round has the structure SubBytes, ShiftRows, MixColumns, AddRoundKey. The standard decryption round has the structure InvShiftRows, InvSubBytes, AddRoundKey, InvMixColumns. Thus, the first two stages of the decryption round need to be interchanged, and the second two changes of the decryption round need to be interchanged.

3.9 Concluding Remarks

Advanced Encryption Standard, or **AES**, is a block cipher adopted as an encryption standard by the US government, and is expected to be used worldwide and analysed extensively, as was the case with its predecessor, the Data Encryption Standard (DES). AES was developed by two Belgian cryptographers, Joan Daemen and Vincent Rijmen, and was based on their previous design, Square. It is also

known by the name of "Rijndael",. AES is not precisely Rijndael, as Rijndael supports a larger range of block and key sizes); AES has a fixed block size of 128 bits and a key size of 128, 192 or 256 bits. Unlike its predecessor DES, it is not a Feistel network, but a substitution-permutation network. AES is fast in both software and hardware, is relatively easy to implement, and requires little memory. As a new encryption standard, it is currently being deployed on a large scale.

As of October 2002, there are no known successful attacks against AES. NSA reviewed all the AES finalists, including Rijndael, and stated that all of them were secure enough for US Government non-classified data. In June 2003, the US Government announced [14] that AES may be used for classified information:

> "The design and strength of all key lengths of the AES algorithm (i.e., 128, 192 and 256) are sufficient to protect classified information up to the SECRET level. TOP SECRET information will require use of either the 192 or 256 key lengths. The implementation of AES in products intended to protect national security systems and/or information must be reviewed and certified by NSA prior to their acquisition and use."

The most common way to attack block ciphers is to try various attacks on versions of the cipher with a reduced number of rounds. AES has 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys. As of 2004, the best known attacks are on 7 rounds for 128-bit keys, 8 rounds for 192-bit keys, and 9 rounds for 256-bit keys [15].

CHAPTER 4

SERIAL IMPLEMENTION OF AES

4.1 Introduction

In the previous chapter, the working principle of AES has been described in details. In order to implement the AES, the main mathematical difficulty with the algorithm is that it uses arithmetic over the field **GF(2^8)**. As some parts of the AES algorithm depend entirely on Finite Fields, the Finite Fields and its different operations in the form of GF(2^8) will be described in details before describing the algorithm itself. At the end of this chapter, some sample input/output will be shown varying the key size and the number of rounds to verify the correctness of the algorithm followed by the run time complexity of AES to defend the parallel implementation of AES.

4.2 Finite Fields

A **field** [16] is an algebraic object with two operations: addition and multiplication, represented by **+** and *****, although they will not necessarily be ordinary addition and multiplication. Using **+**, all the elements of the field must form a commutative group, with identity denoted by **0** and the inverse of **a** denoted by **-a**. Using *****, all the elements of the field except **0** must form another commutative group with identity denoted **1** and inverse of **a** denoted by $\mathbf{a^{-1}}$. (The element **0** has no inverse under *****.) Finally, the **distributive identity** must hold: **a*(b + c) = (a*b) + (a*c)**, for all field elements **a**, **b**, and **c**.

There are a number of different infinite fields, including the rational numbers (fractions), the real numbers (all decimal expansions), and the complex numbers. **Cryptography** focuses on **finite** fields. It turns out that for any prime integer **p** and any integer **n** greater than or equal to **1**, there is a unique field with $\mathbf{p^n}$ elements in it, denoted **GF($\mathbf{p^n}$)**. (The "GF'' stands for "Galois Field'', named after the brilliant

young French mathematician who discovered them.) Here "unique'' means that any two fields with the same number of elements must be essentially the same, except perhaps for giving the elements of the field different names.

In case **n** is equal to **1**, the field is just the integers mod **p**, in which addition and multiplication are just the ordinary versions followed by taking the remainder on division by **p**. The only difficult part of this field is finding the multiplicative inverse of an element, that is, given a non-zero element **a** in $\mathbf{Z_p}$, finding $\mathbf{a^{-1}}$. This is the same as finding a **b** such that **a*b % p = 1**. This calculation can be done with the extended Euclidean algorithm.

4.2.1 The Finite Field GF(2^8)

The case in which **n** is greater than one is much more difficult to describe. In cryptography, **p = 2** and **n = 8**, that is. **GF(2^8)**, because this is the field used by the **Advanced Encryption Standard (AES).**

The AES works primarily with *bytes* (8 bits), represented from the right as:

$$\mathbf{b_7b_6b_5b_4b_3b_2b_1b_0}.$$

The 8-bit elements of the field are regarded as polynomials with coefficients in the field $\mathbf{Z_2}$:

$$\mathbf{b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x^1 + b_0}.$$

The field elements will be denoted by their sequence of bits, using two hex digits.

4.2.2 Addition in GF (2^8)

To add two field elements [16], the corresponding polynomial coefficients are added using addition in $\mathbf{Z_2}$. Here addition is modulo **2**, so that **1 + 1 = 0**, and addition, subtraction and exclusive-or are all the same. The identity element is just zero: **00000000** (in bits) or **0x00** (hex).

4.2.3 Multiplication in GF (2^8)

Multiplication [16] in this field is much more difficult and harder to understand, but it can be implemented very efficiently in hardware and software. The first step in multiplying two field elements is to multiply their corresponding polynomials just as in beginning algebra (except that the coefficients are only **0** or **1**, and **1 + 1 = 0** makes the calculation easier, since many terms just drop out). The result would be up to a degree **14** polynomial -- too big to fit into one byte. A finite field makes use of a fixed degree eight irreducible polynomial (a polynomial that cannot be factored into the product of two simpler polynomials). For the AES, the polynomial used is the following (other polynomials could have been used):

$m(x) = x^8 + x^4 + x^3 + x + 1$ = 0x11b (hex).

The intermediate product of the two polynomials must be divided by **m(x)**. The remainder from this division is the desired product.

This sounds hard, but it is easier to do by hand than it might seem. To make it easier to write the polynomials down, it is required to adopt the convention that instead of **$x^8 + x^4 + x^3 + x + 1$** just the exponents of each non-zero term are written. So for **m(x)**: the exponents of each non-zero term are **(8 4 3 1 0)**.

In order to get the product **(7 5 4 2 1) * (6 4 1 0)** (which is the same as **0xb6 * 0x53** in hexadecimal.) Multiplication will be done first, remembering that in the sum below only an odd number of like powered terms results in a final term:

```
(7 5 4 2 1) * (6 4 1 0)      gives (one term at a time)

(7 5 4 2 1) * (6)    =   (13    11 10 9 8 7)
(7 5 4 2 1) * (4)    =         (11    9 8   6 5)
(7 5 4 2 1) * (1)    =                 (8   6 5   3 2)
(7 5 4 2 1) * (0)    =    +               7   5 4   2 1)
                         ---------------------------------
                         (13       10 9 8     5 4 3   1)
```

The final answer requires the remainder on division by **m(x)**, or **(8 4 3 1)**.

```
                              (13       10 9 8     5 4 3   1)
 (8 4 3 1 0) * (5)     =      (13          9 8   6 5)
                              ---------------------------------
                                        (10       6   4 3   1)
 (8 4 3 1 0) * (2) =                    (10       6 5   3 2)
                              ---------------------------------
                                                   (5 4   2 1)
                                                 (the remainder)
```

Thus the final result of **0xb6 * 0x53 = 0x36** in the field.

4.2.4 Improved Multiplication in GF (2^8)

The above calculations could be converted to a program, but there is a better way. One does the calculations working from the low order terms, and repeatedly multiplying by **(1)** [17]. If the result is of degree **8**, just add (the same as subtract) **m(x)** to get degree **7**. Again this can be illustrated using the above notation and the same example operands: **r * s= (7 5 4 2 1)*(6 4 1 0)**

```
i    powers of  r: r * (i)     Simplified Result          Final Sum
=============================================================================
0                               ( 7 5 4 2 1 )           (7 5 4  2 1 )
-----------------------------------------------------------------------------
1   (7 5 4 2 1 ) * (1)  =  (8 6 5 3 2)
                          +(8 4 3 1 0)
                            ---------------------
                           (6 5 4 2 1 0)   +  (6 5 4 2 1 0)
                                                -------------------
                                                 (7 6           0)
-----------------------------------------------------------------------------
2   (6 5 4   2 1 0) * (1)  =  (7 6 5   3 2 1)
-----------------------------------------------------------------------------
3   (7 6 5   3 2 1) *(1)  =  (8 7 6   4 3 2)
                             +(8       4 3   1 0)
                             ---------------------
                                (7 6       2 1 0)
-----------------------------------------------------------------------------
4   (7 6       2 1 0)*(1)  =  (8 7       3 2 1)
                             +(8       4 3   1 0)
                             ---------------------     (7 6           0)
                                (7     4   2   0)    + (7     4   2   0)
                                                     -------------------
                                                        (6   4   2)
-----------------------------------------------------------------------------
5   (7     4   2   0)*(1)  =  (8     5   3   1)
                             +(8       4 3   1 0)
                             ---------------------
                                    (5 4       0)
                                                        (6   4   2)
6       (5 4       0)*(1)  =      (6 5       1)      +   (6 5       1)
                                                       -----------------
                                                            (5 4   2 1)
```

The final answer is the same as before. Here is an algorithm that realizes the above calculations:

```
byte FFMul(unsigned byte a, unsigned byte b) {
        unsigned byte aa = a, bb = b, r = 0, t;
        while (aa != 0) {
                if ((aa & 1) != 0)
                     r = r ⊕ bb;
                t = bb & 0x80;
                bb = bb << 1; // << denotes left shift operator
                 if (t != 0)
                     bb = bb ⊕ 0x1b;
                aa = aa >> 1; // >> denotes right shift operator
          }
          return r;
}
```

4.3 Algorithm for Multiplicative Inverse of the Form GF(2^8)

In finite fields, it is possible that the harder multiplication can be replaced by the easier addition, at the cost of looking up "logarithms'' and "anti-logarithms''. This fast multiplication algorithm can also be used to find the multiplicative inverse [18] of the form GF(2^8).

From the concept of generator, it is found that **0x03**, which is the same as **x + 1** as a polynomial, is the simplest generator for **GF(2^8)**. Its powers take on all **255** non-zero values of the field. In fact the table below of "exponentials'' or "anti-logs'' gives each possible power. (The table is really just a simple linear table, not really 2-dimensional, but it has been arranged so that the two hex digits are on different axes.) Here **E(rs)** is the field element given by **03^{rs}**, where these are hex numbers, and the initial ``**0x**'' is left off for simplicity.

Table 4.1 Table of exponentials

Table of ``Exponentials'': E(rs) = 03^rs																
E(rs)		**S**														
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
	0	01	03	05	0F	11	33	55	FF	1A	2E	72	96	A1	F8	13	35
	1	5F	E1	38	48	D8	73	95	A4	F7	02	06	0A	1E	22	66	AA
	2	E5	34	5C	E4	37	59	EB	26	6A	BE	D9	70	90	AB	E6	31
	3	53	F5	04	0C	14	3C	44	CC	4F	D1	68	B8	D3	6E	B2	CD
	4	4C	D4	67	A9	E0	3B	4D	D7	62	A6	F1	08	18	28	78	88
	5	83	9E	B9	D0	6B	BD	DC	7F	81	98	B3	CE	49	DB	76	9A
	6	B5	C4	57	F9	10	30	50	F0	0B	1D	27	69	BB	D6	61	A3
r	**7**	FE	19	2B	7D	87	92	AD	EC	2F	71	93	AE	E9	20	60	A0
	8	FB	16	3A	4E	D2	6D	B7	C2	5D	E7	32	56	FA	15	3F	41
	9	C3	5E	E2	3D	47	C9	40	C0	5B	ED	2C	74	9C	BF	DA	75
	A	9F	BA	D5	64	AC	EF	2A	7E	82	9D	BC	DF	7A	8E	89	80
	B	9B	B6	C1	58	E8	23	65	AF	EA	25	6F	B1	C8	43	C5	54
	C	FC	1F	21	63	A5	F4	07	09	1B	2D	77	99	B0	CB	46	CA
	D	45	CF	4A	DE	79	8B	86	91	A8	E3	3E	42	C6	51	F3	0E
	E	12	36	5A	EE	29	7B	8D	8C	8F	8A	85	94	A7	F2	0D	17
	F	39	4B	DD	7C	84	97	A2	FD	1C	24	6C	B4	C7	52	F6	01

Similarly, here is a table of "logarithms'', where the entry **L(rs)** is the field element that satisfies **rs = 03$^{L(rs)}$**, where these are hex numbers, and again the initial ``**0x**'' is left off.

Table 4.2 Table of logarithms

Table of ``logarithms'': rs = 03^L(rs)																	
L(rs)		**S**															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
	0		00	19	01	32	02	1A	C6	4B	C7	1B	68	33	EE	DF	03
	1	64	04	E0	0E	34	8D	81	EF	4C	71	08	C8	F8	69	1C	C1
	2	7D	C2	1D	B5	F9	B9	27	6A	4D	E4	A6	72	9A	C9	09	78
	3	65	2F	8A	05	21	0F	E1	24	12	F0	82	45	35	93	DA	8E
	4	96	8F	DB	BD	36	D0	CE	94	13	5C	D2	F1	40	46	83	38
	5	66	DD	FD	30	BF	06	8B	62	B3	25	E2	98	22	88	91	10
	6	7E	6E	48	C3	A3	B6	1E	42	3A	6B	28	54	FA	85	3D	BA
	7	2B	79	0A	15	9B	9F	5E	CA	4E	D4	AC	E5	F3	73	A7	57
	8	AF	58	A8	50	F4	EA	D6	74	4F	AE	E9	D5	E7	E6	AD	E8
	9	2C	D7	75	7A	EB	16	0B	F5	59	CB	5F	B0	9C	A9	51	A0
	A	7F	0C	F6	6F	17	C4	49	EC	D8	43	1F	2D	A4	76	7B	B7
	B	CC	BB	3E	5A	FB	60	B1	86	3B	52	A1	6C	AA	55	29	9D
	C	97	B2	87	90	61	BE	DC	FC	BC	95	CF	CD	37	3F	5B	D1
	D	53	39	84	3C	41	A2	6D	47	14	2A	9E	5D	56	F2	D3	AB
	E	44	11	92	D9	23	20	2E	89	B4	7C	B8	26	77	99	E3	A5
	F	67	4A	ED	DE	C5	31	FE	18	0D	63	8C	80	C0	F7	70	07

These tables are created using the multiply function in the previous section.

```
void GenerateE() {
      byte x = 0x01;
      int index = 0;
      E[index++] = 0x01;
      for (int i = 0; i < 255; i++) {
          byte y = FFMul(x, 0x03);
          E[index++] = y;
          x = y;
      }
}
void GenerateL() {
             int index;
             for (int i = 0; i < 255; i++) {
                L[E[i] & 0xff] = i;
             }
          }
```

As an example, to calculate the product **b6 * 53** (the same product as in the examples above, leaving off the ``**0x**''). The **L** table above will be used to look up **b6** and **53**: **L(b6) = b1** and **L(53) = 30**. This means that

$$(b6)*(53) = (03)^{(b1)} * (03)^{(30)}$$

$$= (03)^{(b1 + 30)} = (03)^{(e1)}.$$

If the sum above gets bigger than **ff**, **255** will be subtracted as shown. This works because the powers of **03** repeat after **255** iterations. Now using the **E** table to look up $(03)^{(e1)}$, which is the answer: **(36)**.

Thus these tables give a much simpler and faster algorithm for multiplication. Algorithm for fast multiplication is shown below:

```
byte FFMulFast(unsigned byte a, unsigned byte b){
          int  t = 0;;
          if (a == 0 || b == 0) return 0;
          t = L[a] + L[b];
          if (t > 255) t = t - 255;
          return E[t];
}
```

The SubByte transformation of AES requires the multiplicative inverse of each field element except **0**, which has no inverse. This inverse is easy to calculate, given the tables above. If **g** is the generator **03** (leaving off the ``**0x**''), then the inverse of $\mathbf{g^{rs}}$ is $\mathbf{g^{ff - rs}}$, so that for example, to find the inverse of **6b**, look up in the ``log'' table to see that $\mathbf{6b = g^{54}}$, so the inverse of **6b** is $\mathbf{g^{ff - 54} = g^{ab}}$, and from the "exponential" table, this is **df**. The following table shows the result of carrying out the above procedure for each non-zero field element.

Table 4.3 Table of multiplicative inverses

Table of multiplicative inverses: rs*inv(rs) = 01																	
inv(rs)		**S**															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
	0	XX	01	8D	F6	CB	52	7B	D1	E8	4F	29	C0	B0	E1	E5	C7
	1	74	B4	AA	4B	99	2B	60	5F	58	3F	FD	CC	FF	40	EE	B2
	2	3A	6E	5A	F1	55	4D	A8	C9	C1	0A	98	15	30	44	A2	C2
	3	2C	45	92	6C	F3	39	66	42	F2	35	20	6F	77	BB	59	19
	4	1D	FE	37	67	2D	31	F5	69	A7	64	AB	13	54	25	E9	09
	5	ED	5C	05	CA	4C	24	87	BF	18	3E	22	F0	51	EC	61	17
	6	16	5E	AF	D3	49	A6	36	43	F4	47	91	DF	33	93	21	3B
	7	79	B7	97	85	10	B5	BA	3C	B6	70	D0	06	A1	FA	81	82
	8	83	7E	7F	80	96	73	BE	56	9B	9E	95	D9	F7	02	B9	A4
	9	DE	6A	32	6D	D8	8A	84	72	2A	14	9F	88	F9	DC	89	9A
	A	FB	7C	2E	C3	8F	B8	65	48	26	C8	12	4A	CE	E7	D2	62
	B	0C	E0	1F	EF	11	75	78	71	A5	8E	76	3D	BD	BC	86	57
	C	0B	28	2F	A3	DA	D4	E4	0F	A9	27	53	04	1B	FC	AC	E6
	D	7A	07	AE	63	C5	DB	E2	EA	94	8B	C4	D5	9D	F8	90	6B
	E	B1	0D	D6	EB	C6	0E	CF	AD	08	4E	D7	E3	5D	50	1E	B3
	F	5B	23	38	34	68	46	03	8C	DD	9C	7D	A0	CD	1A	41	1C

4.4 Algorithm for Serial Implementation of AES

AES is an iterated block cipher, meaning that the initial input block and cipher key undergoes multiple rounds of transformation before producing the output. Each intermediate cipher result is called a **State.** The block and cipher key are often represented as an array of columns where each array has 4 rows and each column represents a single byte (8 bits). The number of columns in an array representing the state or cipher key, then, can be calculated as the block or key length divided by 32 (32 bits = 4 bytes). An array representing a State will have **Nb** columns, where **Nb** values of 4, 6, and 8 correspond to a 128-, 192-, and 256-bit block, respectively. Similarly, an array representing a Cipher Key will have **Nk** columns, where **Nk** values of 4, 6, and 8 correspond to a 128-, 192-, and 256-bit key, respectively. The AES cipher itself has three operational stages:

1. AddRound Key transformation
2. **Nr**-1 Rounds comprising:
    - SubBytes transformation
    - ShiftRows transformation
    - MixColumns transformation
    - AddRoundKey transformation
3. A final Round comprising:
    - SubBytes transformation
    - ShiftRows transformation
    - AddRoundKey transformation

The overall structure of AES cipher is described below:

Constants: **int Nb = 4;**

int Nr = 10, **12**, or **14;** // rounds, for Nk = 4, 6, or 8

Inputs: array **in** of **4*Nb** bytes // input plaintext

array **out** of **4*Nb** bytes // output ciphertext

array **w** of **4*Nb*(Nr+1)** bytes // expanded key

Internal work array: **state**, 2-dim array of **4*Nb** bytes, **4** rows and **Nb** cols

Algorithm:

```
void Cipher(byte[] in, byte[] out, byte[] w) {
        byte[][] state = new byte[4][Nb];
        state = in;
        AddRoundKey(state, w, 0, Nb - 1); // see Section 4 below
        for (int round = 1; round < Nr; round++) {
                SubBytes(state);
                ShiftRows(state);
                MixColumns(state);
                AddRoundKey(state, w, round*Nb, (round+1)*Nb - 1);
        }
        SubBytes(state);
        ShiftRows(state);
        AddRoundKey(state, w, Nr*Nb, (Nr+1)*Nb - 1);
         out = state;
}
```

In the above algorithm:

- *in[]* and *out[]* are 16-byte arrays with the plaintext and cipher text, respectively. Both of these arrays are actually 4***Nb** bytes in length but **Nb**=4 in AES.
- *state[]* is a 2-dimensional array containing bytes in 4 rows and 4 columns. This arrays is 4 rows by **Nb** columns.
- *w[]* is an array containing the key material and is 4*(**Nr**+1) words in length.
- *AddRoundKey()*, *SubBytes()*, *ShiftRows()*, and *MixColumns()* are functions representing the individual transformations.

4.4.1 The SubBytes Transformation

The substitute bytes transformation operates on each of the State bytes independently and changes the byte value. An S-box, or substitution table, controls

the transformation. One way to think of the SubBytes transformation is that a given byte in State s is given a new value in State s' according to the S-box. The S-box is a function on a byte in State s so that:

$$s'_{i,j} = \text{S-box}\ (s_{i,j})$$

Algorithm for generating the S-Box is as follows:

```
S_Box(byte[16][16] S_Box_Entry) {
        int value = 0;
        For( row = 0; row < 16; row++)
                For (column = 0;column <16;column++){
                        S_Box_Entry[row][column] = value;
                        value++;
                        SubBytesEntry(S_Box_Entry[row][column])
                }
}
byte SubBytesEntry(byte b) {
          byte c = 0x63;
          if (b != 0)  // leave 0 unchanged
                b = multiplicativeInverse(b);
          for (i = 0; i < 8; i++)
                 bi = bi ⊕ b(i+4)%8 ⊕ b(i+5)%8 ⊕ b(i+6)%8 ⊕ b(i+7)%8 ⊕ ci;
          return b;
}
```

Algorithm for **SubByte** transformation is as follows:

```
void SubBytes(byte[][] state) {
  for (int row = 0; row < 4; row++)
    for (int col = 0; col < Nb; col++)
      state[row][col] = Sbox[state[row][col]];
}
```

4.4.2 The ShiftRows Transformation

The shift rows transformation cyclically shifts the bytes in the bottom three rows of the State array. According to the more general Rijndael specification, rows 2, 3, and 4 are cyclically left-shifted by C1, C2, and C3 bytes, respectively, where C1=1,

C2=2, and C3=3. Algorithm for performing the above operations on the state matrix is as follows:

```
void ShiftRows(byte[][] state) {
        byte[] t = new byte[4];
        for (int r = 1; r < 4; r++) {
            for (int c = 0; c < Nb; c++)
                t[c] = state[r][(c + r)%Nb];
            for (int c = 0; c < Nb; c++)
                state[r][c] = t[c];
        }
}
```

4.4.3 The MixColumns Transformation

The action of mixing columns works on the columns of the **state** array, but it is much more complicated than the shift columns action. This transformation treats each column as a four-term polynomial with coefficients in the field **GF(2^8)**. These polynomials are added and multiplied just using the operations of the field **GF(2^8)** on the coefficients, except that the result of a multiplication, which is a polynomial of degree up to **6**, must be reduced by dividing by the polynomial $\mathbf{x^4 + 1}$ and taking the remainder. The columns are each multiplied by the fixed polynomial:

$$\mathbf{a(x) = (03)x^3 + (01)x^2 + (01)x + (02),}$$

where **(03)** stands for the field element **0x03**. The algorithm for mix column transformation is as follows:

```
void MixColumns(byte[][] s) {
  byte[] sp = new byte[4];
  for (int c = 0; c < 4; c++) {
        sp[0] = (0x02 * s[0][c]) ⊕ (0x03 * s[1][c]) ⊕ s[2][c] ⊕ s[3][c];
        sp[1] = s[0][c] ⊕ (0x02 * s[1][c]) ⊕ (0x03 * s[2][c]) ⊕ s[3][c];
        sp[2] = s[0][c] ⊕ s[1][c] ⊕ (0x02 * s[2][c]) ⊕ (0x03 * s[3][c]);
        sp[3] = (0x03 * s[0][c]) ⊕ s[1][c] ⊕ s[2][c] ⊕ (0x02 * s[3][c]);
        for (int i = 0; i < 4; i++)
            s[i][c] = sp[i];
  }
}
```

4.4.4 The AddRoundKey

As described before, portions of the expanded key **w** are exclusive-ored onto the state matrix **Nr+1** times (once for each round plus one more time). There are **4*Nb** bytes of state, and since each byte of the expanded key is used exactly once, the expanded key size of **4*Nb*(Nr+1)** bytes is just right. This algorithm assumes that the key has already been expanded into the array **w**, and it assumes a global counter **wCount** initialized to **0**. The function **AddRoundKey** uses up **4*Nb = 16** bytes of expanded key every time it is called.

```
void AddRoundKey(byte[][] state) {
        for (int c = 0; c < Nb; c++)
                for (int r = 0; r < 4; r++)
                        state[r][c] = state[r][c] ⊕ w[wCount++];
}
```

Algorithm for KeyExpansion is shown below:

```
void KeyExpansion(byte[] key, word[] w, int Nw) {
        int temp;
        int i = 0;

        while ( i < Nk) {
                w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3]);
                i++;
         }
        i = Nk;
        while(i < Nb*(Nr+1)) {
                temp = w[i-1];
                if (i % Nk == 0)
                        temp = SubWord(RotWord(temp)) ⊕ Rcon[i/Nk];
                else if (Nk > 6 && (i%Nk) == 4)
                        temp = SubWord(temp);
                w[i] = w[i-Nk] ⊕ temp;
                 i++;
        }
}
```

4.5 Sample Input/Output

128-bit data, 128-bit key
Encrypting ...

Plaintext	**32**	**43**	**F6**	**A8**	**88**	**5A**	**30**	**8D**	**31**	**31**	**98**	**A2**	**E0**	**37**	**07**	**34**
Key	**2B**	**7E**	**15**	**16**	**28**	**AE**	**D2**	**A6**	**AB**	**F7**	**15**	**88**	**09**	**CF**	**4F**	**3C**
Round1:	19	3D	E3	BE	A0	F4	E2	2B	9A	C6	8D	2A	E9	F8	48	08
Round2:	A4	9C	7F	F2	68	9F	35	2B	6B	5B	EA	43	02	6A	50	49
Round3:	AA	8F	5F	03	61	DD	E3	EF	82	D2	4A	D2	68	32	46	9A
Round4:	48	6C	4E	EE	67	1D	9D	0D	4D	E3	B1	38	D6	5F	58	E7
Round5:	E0	92	7F	E8	C8	63	63	C0	D9	B1	35	50	85	B8	BE	01
Round6:	F1	00	6F	55	C1	92	4C	EF	7C	C8	8B	32	5D	B5	D5	0C
Round7:	26	0E	2E	17	3D	41	B7	7D	E8	64	72	A9	FD	D2	8B	25
Round8:	5A	41	42	B1	19	49	DC	1F	A3	E0	19	65	7A	8C	04	0C
Round9:	EA	83	5C	F0	04	45	33	2D	65	5D	98	AD	85	96	B0	C5
Round10:	EB	40	F2	1E	59	2E	38	84	8B	A1	13	E7	1B	C3	42	D2
Ciphertext	**39**	**25**	**84**	**1D**	**02**	**DC**	**09**	**FB**	**DC**	**11**	**85**	**97**	**19**	**6A**	**0B**	**32**

Decrypting ...

Cipher:	**39**	**25**	**84**	**1D**	**02**	**DC**	**09**	**FB**	**DC**	**11**	**85**	**97**	**19**	**6A**	**0B**	**32**
Key:	**2B**	**7E**	**15**	**16**	**28**	**AE**	**D2**	**A6**	**AB**	**F7**	**15**	**88**	**09**	**CF**	**4F**	**3C**
Round1:	E9	31	7D	B5	CB	32	2C	72	3D	2E	89	5F	AF	09	07	94
Round2:	87	6E	46	A6	F2	4C	E7	8C	4D	90	4A	D8	97	EC	C3	95
Round3:	BE	3B	D4	FE	D4	E1	F2	C8	0A	64	2C	C0	DA	83	86	4D
Round4:	F7	83	40	3F	27	43	3D	F0	9B	B5	31	FF	54	AB	A9	D3
Round5:	A1	4F	3D	FE	78	E8	03	FC	10	D5	A8	DF	4C	63	29	23
Round6:	E1	FB	96	7C	E8	C8	AE	9B	35	6C	D2	BA	97	4F	FB	53
Round7:	52	A4	C8	94	85	11	6A	28	E3	CF	2F	D7	F6	50	5E	07
Round8:	AC	C1	D6	B8	EF	B5	5A	7B	13	23	CF	DF	45	73	11	B5
Round9:	49	DB	87	3B	45	39	53	89	7F	02	D2	F1	77	DE	96	1A
Round10:	D4	BF	5D	30	E0	B4	52	AE	B8	41	11	F1	1E	27	98	E5
Plaintext:	**32**	**43**	**F6**	**A8**	**88**	**5A**	**30**	**8D**	**31**	**31**	**98**	**A2**	**E0**	**37**	**07**	**34**

128-bit data, 192-bit key
Encrypting ...

Plaintext:	**32**	**43**	**F6**	**A8**	**88**	**5A**	**30**	**8D**	**31**	**31**	**98**	**A2**	**E0**	**37**	**07**	**34**
Key:	**2B**	**7E**	**15**	**16**	**28**	**AE**	**D2**	**A6**	**AB**	**F7**	**15**	**88**	**09**	**CF**	**4F**	**3C**
	76	**2E**	**71**	**60**	**F3**	**8B**	**4D**	**A5**								
Round1:	19	3D	E3	BE	A0	F4	E2	2B	9A	C6	8D	2A	E9	F8	48	08
Round2:	72	48	F0	85	13	40	54	3F	5F	65	C0	61	17	35	E7	F1
Round3:	14	E2	0A	1F	B3	DC	3A	62	36	27	2F	D3	DA	75	6F	70
Round4:	CB	42	FD	92	33	3F	28	43	21	11	FE	84	3C	BC	A8	1A
Round5:	94	99	C6	EE	B9	78	94	12	BB	04	09	B7	A7	97	C0	25
Round6:	8A	6C	1E	3E	DB	78	A6	4E	F5	DB	78	62	EA	D6	A4	01
Round7:	43	5C	E2	58	97	7C	16	D8	71	7C	0F	F7	79	19	E5	19
Round8:	70	B8	37	B9	AE	FC	8B	BC	5C	D2	AB	A5	CC	56	D7	4E
Round9:	94	A2	C3	31	ED	28	BF	DE	D7	D6	C5	83	4B	A9	ED	1E
Round10:	52	2D	88	C5	ED	AB	19	4E	25	EC	73	1C	11	FA	6B	08
Round11:	AB	82	54	06	DA	72	4D	0C	2B	CC	F6	C2	39	32	12	01
Round12:	43	88	B3	26	6A	F7	68	E8	4F	CC	A4	2A	3A	4D	45	5F
Ciphertext:	**F9**	**FB**	**29**	**AE**	**FC**	**38**	**4A**	**25**	**03**	**40**	**D8**	**33**	**B8**	**7E**	**BC**	**00**

Decrypting ...

Ciphertext:	**F9**	**FB**	**29**	**AE**	**FC**	**38**	**4A**	**25**	**03**	**40**	**D8**	**33**	**B8**	**7E**	**BC**	**00**
Key:	**2B**	**7E**	**15**	**16**	**28**	**AE**	**D2**	**A6**	**AB**	**F7**	**15**	**88**	**09**	**CF**	**4F**	**3C**
	76	**2E**	**71**	**60**	**F3**	**8B**	**4D**	**A5**								
Round1:	1A	68	49	CF	02	4B	6E	F7	84	E3	6D	9B	80	C4	45	E5
Round2:	62	40	42	7C	57	4B	C9	6F	F1	23	20	FE	12	13	E3	25
Round3:	00	62	8F	30	55	CE	7F	A6	3F	2D	C4	2F	82	D8	D4	9C
Round4:	22	34	A6	72	55	F6	55	C7	0E	D3	2E	1D	B3	3A	08	EC
Round5:	51	B0	62	2F	E4	B5	0E	56	4A	B1	9A	65	4B	6C	3D	06
Round6:	1A	10	76	D4	88	10	D9	6A	A3	D4	98	61	B6	4A	47	68
Round7:	7E	BC	BC	7C	B9	B9	49	B2	E6	F6	72	2F	87	50	24	AA
Round8:	22	BC	01	3F	56	F2	BA	28	EA	88	B4	C9	5C	EE	22	A9
Round9:	1F	75	BB	A2	C3	82	C2	4F	FD	65	54	1A	EB	2C	34	5F
Round10:	FA	86	15	51	6D	CC	A8	C0	05	9D	67	AA	57	98	80	66
Round11:	40	09	BA	A1	7D	4D	94	97	CF	96	8C	75	F0	52	20	EF
Round12:	D4	BF	5D	30	E0	B4	52	AE	B8	41	11	F1	1E	27	98	E5
Plaintext:	**32**	**43**	**F6**	**A8**	**88**	**5A**	**30**	**8D**	**31**	**31**	**98**	**A2**	**E0**	**37**	**07**	**34**

128-bit data, 256-bit key
Encrypting ...

Plaintext:	**32**	**43**	**F6**	**A8**	**88**	**5A**	**30**	**8D**	**31**	**31**	**98**	**A2**	**E0**	**37**	**07**	**34**
Key:	**2B**	**7E**	**15**	**16**	**28**	**AE**	**D2**	**A6**	**AB**	**F7**	**15**	**88**	**09**	**CF**	**4F**	**3C**
	76	**2E**	**71**	**60**	**F3**	**8B**	**4D**	**A5**	**6A**	**78**	**4D**	**90**	**45**	**19**	**0C**	**FE**
Round1:	19	3D	E3	BE	A0	F4	E2	2B	9A	C6	8D	2A	E9	F8	48	08
Round2:	72	48	F0	85	13	40	54	3F	22	80	9E	EA	6D	1F	2A	B2
Round3:	59	F8	A8	D4	12	71	BF	44	E2	2B	A6	5E	5D	69	9A	49
Round4:	88	A8	EB	D5	66	49	40	5E	9F	AD	55	E9	33	0D	7F	84
Round5:	6D	0C	80	51	D5	BC	1D	B5	C5	1F	45	0F	18	46	7F	34
Round6:	CF	F5	04	43	27	6F	76	55	A5	5A	FD	7B	B6	99	F4	5F
Round7:	63	93	D6	68	2D	DA	2E	4F	42	88	77	37	12	57	8A	11
Round8:	29	26	AE	58	F4	32	23	4B	F0	70	FF	6E	56	9E	44	23
Round9:	3F	DA	E4	32	0E	CE	55	CE	C9	32	D6	55	E4	3A	CD	2B
Round10:	D6	40	90	18	38	64	11	35	61	EF	7C	37	99	00	31	FD
Round11:	6B	3C	E6	72	EA	E1	1D	52	E2	8F	1D	54	96	E0	C0	D0
Round12:	84	74	88	72	49	E5	0A	9F	17	C0	5A	37	A1	A6	9F	41
Round13:	BF	8A	29	14	80	F8	06	21	44	3E	2B	81	AA	2F	4C	16
Round14:	D3	20	3D	D1	1A	C7	2D	8B	5E	C4	72	24	95	3D	FE	5B
Ciphertext:	**1A**	**6E**	**6C**	**2C**	**66**	**2E**	**7D**	**A6**	**50**	**1F**	**FB**	**62**	**BC**	**9E**	**93**	**F3**

Decrypting ...

Ciphertext:	**1A**	**6E**	**6C**	**2C**	**66**	**2E**	**7D**	**A6**	**50**	**1F**	**FB**	**62**	**BC**	**9E**	**93**	**F3**
Key:	**2B**	**7E**	**15**	**16**	**28**	**AE**	**D2**	**A6**	**AB**	**F7**	**15**	**88**	**09**	**CF**	**4F**	**3C**
	76	**2E**	**71**	**60**	**F3**	**8B**	**4D**	**A5**	**6A**	**78**	**4D**	**90**	**45**	**19**	**0C**	**FE**
Round1:	66	C6	40	39	A2	1C	BB	3E	58	27	27	3D	2A	B7	D8	36
Round2:	08	41	F1	47	CD	B2	29	FA	1B	15	A5	FD	AC	7E	6F	0C
Round3:	5F	D9	BE	83	3B	BA	DB	40	F0	24	C4	DB	32	92	67	9A
Round4:	7F	F8	A4	70	87	73	BA	40	98	E1	8E	00	90	EB	A4	20
Round5:	F6	43	10	54	07	DF	C7	AD	EF	63	60	96	EE	09	82	9A
Round6:	75	8B	F6	F1	AB	23	BD	23	DD	80	69	8B	69	57	FC	FC
Round7:	A5	23	16	26	BF	51	1B	6A	8C	0B	E4	B3	B1	F7	26	9F
Round8:	FB	57	F5	82	D8	C4	7E	45	2C	5B	F6	84	C9	DC	31	9A
Round9:	8A	A8	54	CF	CC	BE	BF	1A	06	EE	F2	FC	4E	E6	38	21
Round10:	3C	65	6E	18	03	C0	D2	D1	A6	5A	CD	D5	AD	FE	A4	76
Round11:	C4	3B	FC	5F	33	95	D2	03	DB	D7	E9	58	C3	C2	09	1E
Round12:	CB	A3	24	3B	C9	F1	B8	48	98	F9	C2	1B	4C	41	08	58
Round13:	40	09	0B	37	7D	CD	E5	97	93	C0	8C	75	3C	52	20	87
Round14:	D4	BF	5D	30	E0	B4	52	AE	B8	41	11	F1	1E	27	98	E5
Plaintext:	**32**	**43**	**F6**	**A8**	**88**	**5A**	**30**	**8D**	**31**	**31**	**98**	**A2**	**E0**	**37**	**07**	**34**

128-bit data, 128-bit key
Encrypting ...

Plaintext:	**00**	**11**	**22**	**33**	**44**	**55**	**66**	**77**	**88**	**99**	**AA**	**BB**	**CC**	**DD**	**EE**	**FF**
Key:	**00**	**01**	**02**	**03**	**04**	**05**	**06**	**07**	**08**	**09**	**0A**	**0B**	**0C**	**0D**	**0E**	**0F**
Round1:	00	10	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0
Round2:	89	D8	10	E8	85	5A	CE	68	2D	18	43	D8	CB	12	8F	E4
Round3:	49	15	59	8F	55	E5	D7	A0	DA	CA	94	FA	1F	0A	63	F7
Round4:	FA	63	6A	28	25	B3	39	C9	40	66	8A	31	57	24	4D	17
Round5:	24	72	40	23	69	66	B3	FA	6E	D2	75	32	88	42	5B	6C
Round6:	C8	16	77	BC	9B	7A	C9	3B	25	02	79	92	B0	26	19	96
Round7:	C6	2F	E1	09	F7	5E	ED	C3	CC	79	39	5D	84	F9	CF	5D
Round8:	D1	87	6C	0F	79	C4	30	0A	B4	55	94	AD	D6	6F	F4	1F
Round9:	FD	E3	BA	D2	05	E5	D0	D7	35	47	96	4E	F1	FE	37	F1
Round10:	BD	6E	7C	3D	F2	B5	77	9E	0B	61	21	6E	8B	10	B6	89
Ciphertext:	**69**	**C4**	**E0**	**D8**	**6A**	**7B**	**04**	**30**	**D8**	**CD**	**B7**	**80**	**70**	**B4**	**C5**	**5A**

Decrypting ..

Ciphertext:	**69**	**C4**	**E0**	**D8**	**6A**	**7B**	**04**	**30**	**D8**	**CD**	**B7**	**80**	**70**	**B4**	**C5**	**5A**
Key:	**00**	**01**	**02**	**03**	**04**	**05**	**06**	**07**	**08**	**09**	**0A**	**0B**	**0C**	**0D**	**0E**	**0F**
Round1:	7A	D5	FD	A7	89	EF	4E	27	2B	CA	10	0B	3D	9F	F5	9F
Round2:	54	D9	90	A1	6B	A0	9A	B5	96	BB	F4	0E	A1	11	70	2F
Round3:	3E	1C	22	C0	B6	FC	BF	76	8D	A8	50	67	F6	17	04	95
Round4:	B4	58	12	4C	68	B6	8A	01	4B	99	F8	2E	5F	15	55	4C
Round5:	E8	DA	B6	90	14	77	D4	65	3F	F7	F5	E2	E7	47	DD	4F
Round6:	36	33	9D	50	F9	B5	39	26	9F	2C	09	2D	C4	40	6D	23
Round7:	2D	6D	7E	F0	3F	33	E3	34	09	36	02	DD	5B	FB	12	C7
Round8:	3B	D9	22	68	FC	74	FB	73	57	67	CB	E0	C0	59	0E	2D
Round9:	A7	BE	1A	69	97	AD	73	9B	D8	C9	CA	45	1F	61	8B	61
Round10:	63	53	E0	8C	09	60	E1	04	CD	70	B7	51	BA	CA	D0	E7
Plaintext:	**00**	**11**	**22**	**33**	**44**	**55**	**66**	**77**	**88**	**99**	**AA**	**BB**	**CC**	**DD**	**EE**	**FF**

128-bit data, 192-bit key
Encrypting ...

Plaintext:	**00**	**11**	**22**	**33**	**44**	**55**	**66**	**77**	**88**	**99**	**AA**	**BB**	**CC**	**DD**	**EE**	**FF**
Key:	**00**	**01**	**02**	**03**	**04**	**05**	**06**	**07**	**08**	**09**	**0A**	**0B**	**0C**	**0D**	**0E**	**0F**
	10	**11**	**12**	**13**	**14**	**15**	**16**	**17**								
Round1:	00	10	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0
Round2:	4F	63	76	06	43	E0	AA	85	AF	F8	C9	D0	41	FA	0D	E4
Round3:	CB	02	81	8C	17	D2	AF	9C	62	AA	64	42	8B	B2	5F	D7
Round4:	F7	5C	77	78	A3	27	C8	ED	8C	FE	BF	C1	A6	C3	7F	53
Round5:	22	FF	C9	16	A8	14	74	41	64	96	F1	9C	64	AE	25	32
Round6:	80	12	1E	07	76	FD	1D	8A	8D	8C	31	BC	96	5D	1F	EE
Round7:	67	1E	F1	FD	4E	2A	1E	03	DF	DC	B1	EF	3D	78	9B	30
Round8:	0C	03	70	D0	0C	01	E6	22	16	6B	8A	CC	D6	DB	3A	2C
Round9:	72	55	DA	D3	0F	B8	03	10	E0	0D	6C	6B	40	D0	52	7C
Round10:	A9	06	B2	54	96	8A	F4	E9	B4	BD	B2	D2	F0	C4	43	36
Round11:	88	EC	93	0E	F5	E7	E4	B6	CC	32	F4	C9	06	D2	94	14
Round12:	AF	B7	3E	EB	1C	D1	B8	51	62	28	0F	27	FB	20	D5	85
Ciphertext:	**DD**	**A9**	**7C**	**A4**	**86**	**4C**	**DF**	**E0**	**6E**	**AF**	**70**	**A0**	**EC**	**0D**	**71**	**91**

Decrypting ..

Ciphertext:	**DD**	**A9**	**7C**	**A4**	**86**	**4C**	**DF**	**E0**	**6E**	**AF**	**70**	**A0**	**EC**	**0D**	**71**	**91**
Key:	**00**	**01**	**02**	**03**	**04**	**05**	**06**	**07**	**08**	**09**	**0A**	**0B**	**0C**	**0D**	**0E**	**0F**
	10	**11**	**12**	**13**	**14**	**15**	**16**	**17**								
Round1:	79	3E	76	97	9C	34	03	E9	AA	B7	B2	D1	0F	A9	6C	CC
Round2:	C4	94	BF	FA	E6	23	22	AB	4B	B5	DC	4E	6F	CE	69	DD
Round3:	D3	7E	37	05	90	7A	1A	20	8D	1C	37	1E	8C	6F	BF	B5
Round4:	40	6C	50	10	76	D7	00	66	E1	70	57	CA	09	FC	7B	7F
Round5:	FE	7C	7E	71	FE	7F	80	70	47	B9	51	93	F6	7B	8E	4B
Round6:	85	E5	C8	04	2F	86	14	54	9E	BC	A1	7B	27	72	72	DF
Round7:	CD	54	C7	28	38	64	C0	C5	5D	4C	72	7E	90	C9	A4	65
Round8:	93	FA	A1	23	C2	90	3F	47	43	E4	DD	83	43	16	92	DE
Round9:	68	CC	08	ED	0A	BB	D2	BC	64	2E	F5	55	24	4A	E8	78
Round10:	1F	B5	43	0E	F0	AC	CF	64	AA	37	0C	DE	3D	77	79	2C
Round11:	84	E1	DD	69	1A	41	D7	6F	79	2D	38	97	83	FB	AC	70
Round12:	63	53	E0	8C	09	60	E1	04	CD	70	B7	51	BA	CA	D0	E7
Plaintext:	**00**	**11**	**22**	**33**	**44**	**55**	**66**	**77**	**88**	**99**	**AA**	**BB**	**CC**	**DD**	**EE**	**FF**

128-bit data, 256-bit key
Encrypting ...

Plaintext:	**00**	**11**	**22**	**33**	**44**	**55**	**66**	**77**	**88**	**99**	**AA**	**BB**	**CC**	**DD**	**EE**	**FF**
Key:	**00**	**01**	**02**	**03**	**04**	**05**	**06**	**07**	**08**	**09**	**0A**	**0B**	**0C**	**0D**	**0E**	**0F**
	10	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**1A**	**1B**	**1C**	**1D**	**1E**	**1F**
Round1:	00	10	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0
Round2:	4F	63	76	06	43	E0	AA	85	EF	A7	21	32	01	A4	E7	05
Round3:	18	59	FB	C2	8A	1C	00	A0	78	ED	8A	AD	C4	2F	61	09
Round4:	97	5C	66	C1	CB	9F	3F	A8	A9	3A	28	DF	8E	E1	0F	63
Round5:	1C	05	F2	71	A4	17	E0	4F	F9	21	C5	C1	04	70	15	54
Round6:	C3	57	AA	E1	1B	45	B7	B0	A2	C7	BD	28	A8	DC	99	FA
Round7:	7F	07	41	43	CB	4E	24	3E	C1	0C	81	5D	83	75	D5	4C
Round8:	D6	53	A4	69	6C	A0	BC	0F	5A	CA	AB	5D	B9	6C	5E	7D
Round9:	5A	A8	58	39	5F	D2	8D	7D	05	E1	A3	88	68	F3	B9	C5
Round10:	4A	82	48	51	C5	7E	7E	47	64	3D	E5	0C	2A	F3	E8	C9
Round11:	C1	49	07	F6	CA	3B	3A	A0	70	E9	AA	31	3B	52	B5	EC
Round12:	5F	9C	6A	BF	BA	C6	34	AA	50	40	9F	A7	66	67	76	53
Round13:	51	66	04	95	43	53	95	03	14	FB	86	E4	01	92	25	21
Round14:	62	7B	CE	B9	99	9D	5A	AA	C9	45	EC	F4	23	F5	6D	A5
Ciphertext:	**8E**	**A2**	**B7**	**CA**	**51**	**67**	**45**	**BF**	**EA**	**FC**	**49**	**90**	**4B**	**49**	**60**	**89**

Decrypting...

Ciphertext:	**8E**	**A2**	**B7**	**CA**	**51**	**67**	**45**	**BF**	**EA**	**FC**	**49**	**90**	**4B**	**49**	**60**	**89**
Key:	**00**	**01**	**02**	**03**	**04**	**05**	**06**	**07**	**08**	**09**	**0A**	**0B**	**0C**	**0D**	**0E**	**0F**
	10	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**1A**	**1B**	**1C**	**1D**	**1E**	**1F**
Round1:	AA	5E	CE	06	EE	6E	3C	56	DD	E6	8B	AC	26	21	BE	BF
Round2:	D1	ED	44	FD	1A	0F	3F	2A	FA	4F	F2	7B	7C	33	2A	69
Round3:	CF	B4	DB	ED	F4	09	38	08	53	85	02	AC	33	DE	18	5C
Round4:	78	E2	AC	CE	74	1E	D5	42	51	00	C5	E0	E2	3B	80	C7
Round5:	D6	F3	D9	DD	A6	27	9B	D1	43	0D	52	A0	E5	13	F3	FE
Round6:	BE	B5	0A	A6	CF	F8	56	12	6B	0D	6A	FF	45	C2	5D	C4
Round7:	F6	E0	62	FF	50	74	58	F9	BE	50	49	76	56	ED	65	4C
Round8:	D2	2F	0C	29	1F	FE	03	1A	78	9D	83	B2	EC	C5	36	4C
Round9:	2E	6E	7A	2D	AF	C6	EE	F8	3A	86	AC	E7	C2	5B	A9	34
Round10:	9C	F0	A6	20	49	FD	59	A3	99	51	89	84	F2	6B	E1	78
Round11:	88	DB	34	FB	1F	80	76	78	D3	F8	33	C2	19	4A	75	9E
Round12:	AD	9C	7E	01	7E	55	EF	25	BC	15	0F	E0	1C	CB	63	95
Round13:	84	E1	FD	6B	1A	5C	94	6F	DF	49	38	97	7C	FB	AC	23
Round14:	63	53	E0	8C	09	60	E1	04	CD	70	B7	51	BA	CA	D0	E7
Plaintext:	**00**	**11**	**22**	**33**	**44**	**55**	**66**	**77**	**88**	**99**	**AA**	**BB**	**CC**	**DD**	**EE**	**FF**

4.6 Run Time Complexity of the Serial Implementation

The number of steps an algorithm requires to solve a specific problem is denoted as the running time of the algorithm. In general, the running time depends on the size of the problem and on the respective input. In order to evaluate an algorithm independently of the input, the notation of time complexity is introduced. The time complexity T(n) is a function of the problem size n. The value of T(n) is the running time of the algorithm in the worst case, i.e. the number of steps it requires at most with an arbitrary input. However, time complexity function does not give the actual execution time of an algorithm rather it gives an idea how the time required for an algorithm changes as the problem size increases.

In order to compute the run time complexity of the AES algorithm, the time complexity function for each transformation has to be considered. As the AES algorithm consists of only four different types of transformation, the time complexity function of AES will depend on the time complexity of each transformation.

Time Complexity of AddRoundKey Transformation:

In the **AddRoundKey** transformation, 128 bits of **state** matrix is XORed with the 128 bits of the round key where 128 bits of state matrix and round key is arranged in 4X4 matrix. So, there are 16 XORed operation is performed. Hence, the time complexity function for AddRoundKey is as follows:

$$T_{ARK}(N) = 11 * 16 * N$$
$$= 176N$$

(AddRoundKey function executes 11 times for encrypting one block of data)

Time Complexity of SubBytes Transformation:

In the **SubBytes transformation** each individual byte of **State** is mapped into a new byte in the following way: The leftmost 4 bits of the byte are used as a row value and the rightmost 4 bits are used as a column value. These row and column

values serve as indexes into the S-box to select a unique 8-bit output value. So, there are again 16 substitution operations are performed. Hence, the time complexity function for **SubBytes** is as follows:

$$T_{SB}(N) = 10 * 16 * N$$

$$= 160N$$

Time Complexity of ShiftRows Transformation:

In the **ShiftRows** transformation, the first row of the state is not altered. For the second row, a 1-byte circular left shift is performed. For the third row, a 2-byte circular left shift is performed. For the fourth row, a 3-byte circular left shift is performed. Hence, the time complexity function for **ShiftRows** is as follows:

$$T_{SR}(N) = (10 * 8 * 4)N$$

$$= 320N$$

Time Complexity of MixColumn Transformation:

MixColumn transformation uses a mathematical function to transform the values of a given column within a State. It operates on each column individually. Each byte of a column is mapped into a new value that is a function of all four bytes in the column. Each column requires 20 addition and multiplication operations in the form of $GF(2^8)$. Therefore, the time complexity function of MixColumn transformation is as follows:

$$T_{MC}(N) = 10 * 4 * (20 + 4)N$$

$$= 960N$$

Therefore, the time complexity function of AES algorithm:

$$T_{AES}(N) = 176N + 160N + 320N + 960N$$

$$= 1616N$$

$$T_{AES}(N) = O(N) \quad \text{[when the value of N is very large]}$$

Performance of AES in a serial machine:

The above AES algorithm is implemented in a serial machine with the following hardware configuration:

Hardware Configuration

Processor **:** Pentium III

Clock Speed : 800 MHz

Ram : 128 MB

Cache : 256 KB

Table 4.4 Performance of AES in a serial machine

Key Size	**Number of Data Blocks**						
128 bit	**5000**	**10000**	**15000**	**20000**	**25000**	**30000**	**50000**
	ms	**ms**	**Ms**	**ms**	**ms**	**ms**	**ms**
Test1	453	906	1359	1797	2250	2687	4500
Test2	469	922	1344	1797	2250	2719	4510
Test3	454	906	1360	1797	2250	2703	4500
Avg.	458	911	1354	1797	2250	2703	4503
Time/ Block	91.6 μs	91.1 μs	90.2 μs	89.8 μs	90.0 μs	90.1 μs	90.0 μs
192 Bit	**5000**	**10000**	**15000**	**20000**	**25000**	**30000**	**50000**
	ms	**ms**	**Ms**	**ms**	**ms**	**ms**	**ms**
Test1	562	1125	1641	2203	2734	3281	5515
Test2	547	1125	1640	2218	2765	3297	5484
Test3	560	1120	1649	2203	2740	3297	5500
Avg.	556	1123	1643	2208	2746	3291	5499
Time/ Block	111.2 μs	112.3 μs	109.5 μs	110.4 μs	109.8 μs	109.7 μs	110.0 μs
256 Bit	**5000**	**10000**	**15000**	**20000**	**25000**	**30000**	**50000**
	ms	**ms**	**Ms**	**ms**	**ms**	**ms**	**ms**
Test1	657	1297	1953	2610	3219	3875	6453
Test2	657	1297	1937	2578	3234	3875	6500
Test3	660	1300	1948	2597	3219	3875	6453
Avg.	658	1298	1946	2595	3224	3875	6468
Time/ Block	131.6 μs	129.8 μs	129.7 μs	129.7 μs	128.9 μs	129.1 μs	129.3 μs

4.7 Summery

From the above time complexity function, it is found that the AES algorithm has a linear complexity that means when the value of N (number of data block) ranges from 10 – 100000, the execution time will vary from 10^{-5} second to 1 seconds (each operation is assumed to take 10^{-6} second). However, when the value of N is greater than 10^8, the execution time of the algorithm will require several days to encrypt or decrypt. The following table will give a clear idea:

Table 4.5 Computer time used for different data blocks

Problem Size (No. of data blocks)	No. of Operations	Time Required
10	10	10^{-5} Sec
10^2	100	10^{-4} Sec
10^3	1000	10^{-3} Sec
10^4	10000	10^{-2} Sec
10^5	100000	10^{-1} Sec
10^6	1000000	1 Sec
10^{10}	10000000000	2.78 hours
10^{12}	1000000000000	11.57 days

Table 4.4 also proves that time complexity function of AES is linear i.e. when the number of data blocks increase; the run time of AES also increases proportionately. To improve the performance of the AES algorithm, parallel implementation of AES can be an alternate choice where several computers will be used in parallel to encrypt or decrypt the data blocks.

CHAPTER 5

PARALLEL IMPLEMENTAION OF AES

5.1 Introduction

The current trend in high performance computing is clustering and distributed computing. In clusters, powerful low cost workstations and/or PCs are linked through fast communication interfaces to achieve high performance parallel computing. Recent increases in communication speeds, microprocessor clock speeds, availability of high performance public domain software including operating system, compiler tools and message passing libraries, make cluster based computing appealing in terms of both high performance computing and cost effectiveness.

Parallel computing on clustered systems is a viable and attractive proposition due to the high communication speeds of modern networks. To efficiently use more than one processor in a program, the processors must share data and co-ordinate access to and updating of the shared data. The most popular approach to this problem is to exchange of data through messages between computers. The MPI (message Passing Interface) approach is considered to be one of the most mature methods currently used in parallel programming mainly due to the relative simplicity of using the method by writing a set of library functions or an API (Application Program Interface) callable from C, C++ or Fortran Programs. MPI was designed for high performance on both massively parallel machines and clusters. Today, MPI is considered a de facto standard for message passing in the parallel-computing paradigm. For implementing the AES algorithm in parallel, the MPI based cluster is used in the present chapter.

The performance of a parallel algorithm depends not only on input size but also on the architecture of the parallel computer, the number of processors, and the interconnection network. In this chapter, different types of parallel architectures and interconnection networks are discussed before actually implementing the parallel

algorithm of AES. At the end of this chapter, some sample input/output are shown varying the key size, number of rounds and the number of processors to verify the correctness of parallel algorithm. Finally, the run time complexity of the parallel algorithm is shown to measure the performance improvement of the parallel implementation over the serial implementation.

5.2 Parallel Architectures

5.2.1 Introduction to SIMD Architectures

SIMD (Single-Instruction Stream Multiple-Data Stream) [19] architectures are essential in the parallel world of computers. In SIMD architectures, several processing elements are supervised by one control unit. All the processing units receive the same instruction from the control unit but operate on different data sets, which come from different data flows, meaning that they execute programs in a lock-step mode, in which each processing element has its own data stream. There are two types of SIMD architectures: the True SIMD and the Pipelined SIMD. Each has its own advantages and disadvantages but their common attribute is superior ability to manipulate vectors.

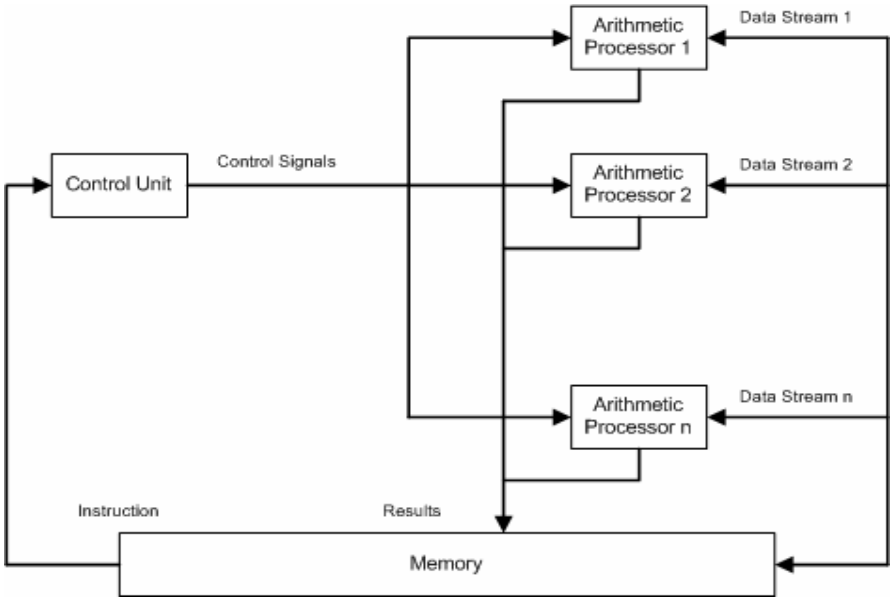


Figure 5.1 Model of an SIMD architecture

SIMD with Distributed Memory

The true SIMD architecture contains a single control unit (CU) with multiple processor elements (PE) acting as arithmetic units (AU). In this situation, the arithmetic units are slaves to the control unit. The AU's cannot fetch or interpret any instructions. They are merely a unit which has capabilities of addition, subtraction, multiplication, and division. Each AU has access only to its own memory. In this sense, if a AU needs the information contained in a different AU, it must put in a request to the CU and the CU must manage the transferring of information. The advantage of this type of architecture is in the ease of adding more memory and AU's to the computer. The disadvantage can be found in the time wasted by the CU managing all memory exchanges.

SIMD with Shared Memory

Another true SIMD architecture is designed with a configurable association between the PE's and the memory modules (M). In this architecture, the local memories that were attached to each AU as above are replaced by memory modules. These M's are shared by all the PE's through an alignment network or switching unit. This allows for the individual PE's to share their memory without accessing the control unit. This type of architecture is certainly superior to the above, but a disadvantage is inherited in the difficulty of adding memory.

Pipelined SIMD

Pipelined SIMD architecture is composed of a pipeline of arithmetic units with shared memory. The pipeline takes different streams of instructions and performs all the operations of an arithmetic unit. The pipeline is a first in first out type of procedure. The size of the pipelines are relative. To take advantage of the pipeline, the data to be evaluated must be stored in different memory modules so the pipeline can be fed with this information as fast as possible. The advantages to this architecture can be

found in the speed and efficiency of data processing assuming the above stipulation is met.

5.2.2 Introduction to MIMD Architectures

Multiple instruction stream, multiple data stream (MIMD) [20] machines have a number of processors that function asynchronously and independently. At any time, different processors may be executing different instructions on different pieces of data. MIMD architectures may be used in a number of application areas such as computer-aided design/computer-aided manufacturing, simulation, modeling, and as communication switches. MIMD machines can be of either shared memory or distributed memory categories. These classifications are based on how MIMD processors access memory. Shared memory machines may be of the bus-based, extended, or hierarchical type. Distributed memory machines may have hypercube or mesh interconnection schemes.

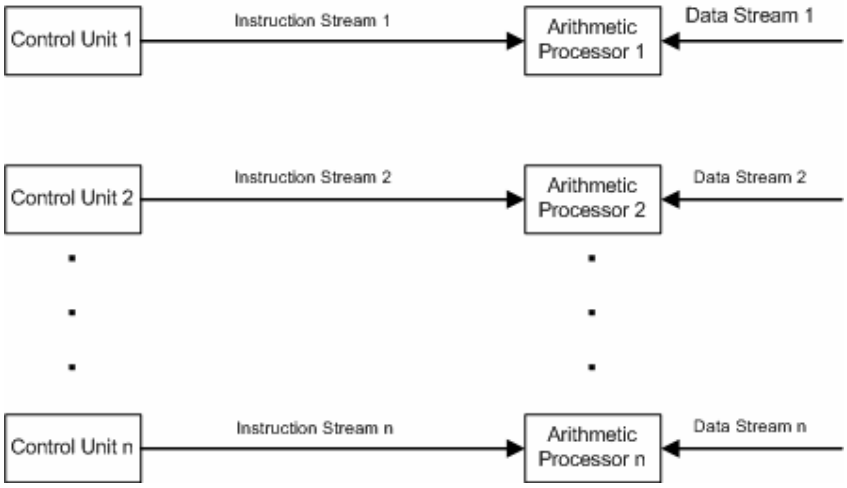


Figure 5.2 Model of an MISD architecture

Shared Memory: Bus-based

MIMD machines with shared memory have processors which share a common, central memory. In the simplest form, all processors are attached to a bus which connects them to memory. This setup is called bus-based shared memory. Bus-based machines may have another bus that enables them to communicate directly with one

another. This additional bus is used for synchronization among the processors. When using bus-based shared memory MIMD machines, only a small number of processors can be supported. There is contention among the processors for access to shared memory, so these machines are limited for this reason. These machines may be incrementally expanded up to the point where there is too much contention on the bus.

Shared Memory: Extended

MIMD machines with extended shared memory attempt to avoid or reduce the contention among processors for shared memory by subdividing the memory into a number of independent memory units. These memory units are connected to the processors by an interconnection network. The memory units are treated as a unified central memory. One type of interconnection network for this type of architecture is a crossbar switching network. In this scheme, N processors are linked to M memory units which require N times M switches. This is not an economically feasible setup for connecting a large number of processors.

Shared Memory: Hierarchical

MIMD machines with hierarchical shared memory use a hierarchy of buses to give processors access to each other's memory. Processors on different boards may communicate through internodal buses. Buses support communication between boards. With this type of architecture, the machine may support over a thousand of processors.

Distributed Memory: Introduction

In distributed memory MIMD machines, each processor has its own individual memory location. For data to be shared, it must be passed from one processor to another as a message. Since there is no shared memory, contention is not as great a problem with these machines. It is not economically feasible to connect a large number of processors directly to each other. A way to avoid this multitude of direct

connections is to connect each processor to just a few others. This type of design can be inefficient because of the added time required to pass a message from one processor to another along the message path. The amount of time required for processors to perform simple message routing can be substantial. Systems were designed to reduce this time loss and hypercube and mesh are among two of the popular interconnection schemes.

Distributed Memory: Hypercube Interconnection Network

In an MIMD distributed memory machine with a hypercube system interconnection network containing four processors, a processor and a memory module are placed at each vertex of a square. The diameter of the system is the minimum number of steps it takes for one processor to send a message to the processor that is the farthest away. So, for example, the diameter of a 2-cube is 2. In a hypercube system with eight processors and each processor and memory module being placed in the vertex of a cube, the diameter is 3. In general, a system that contains 2^N processors with each processor directly connected to N other processors, the diameter of the system is N. One disadvantage of a hypercube system is that it must be configured in powers of two, so a machine must be built that could potentially have many more processors than is really needed for the application.

Distributed Memory: Mesh Interconnection Network

In an MIMD distributed memory machine with a mesh interconnection network, processors are placed in a two-dimensional grid. Each processor is connected to its four immediate neighbors. Wraparound connections may be provided at the edges of the mesh. One advantage of the mesh interconnection network over the hypercube is that the mesh system need not be configured in powers of two. A disadvantage is that the diameter of the mesh network is greater than the hypercube for systems with more than four processors.

For implementing the AES algorithm in parallel, the SIMD architecture with star interconnection network has been used. The reason for using the SIMD architecture is that the algorithm has been designed in a way called data parallelism where the data are divided among the processors by decomposing the data into tasks that can be assigned to different processors. This design of algorithm is best suited with SIMD architecture.

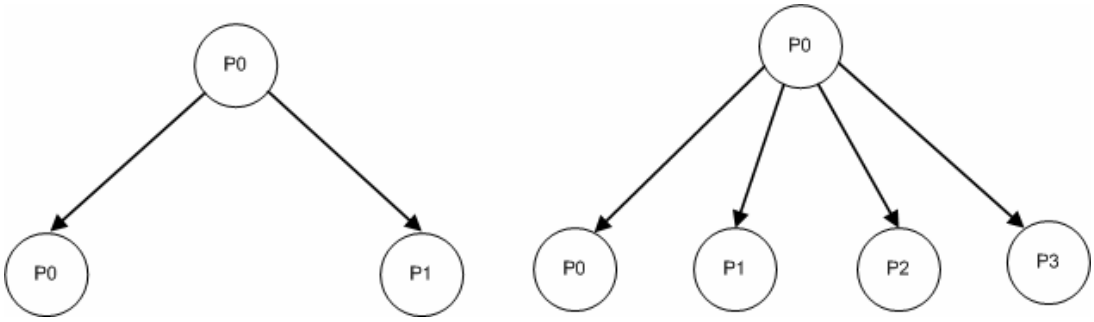
5.3 Algorithm for Parallel Implementation of AES

There are two major components of parallel algorithm design. The first one is the identification and specification of the overall problem as a set of tasks that can be performed concurrently. The second is the mapping of these tasks onto different processors so that the overall communication overhead is minimized. The first component specifies concurrency, and the second one specifies data locality. The performance of an algorithm on a parallel architecture depends on both. Concurrency is necessary to keep the processors busy. Locality is important because it minimizes communication overhead. Ideally, a parallel algorithm should have maximum concurrency and locality. However, for most algorithms, there is a tradeoff. An algorithm that has more concurrency often has less locality.

To implement the AES algorithm in parallel, data blocks (Figure 5.3) and a key are distributed among the available processors. Each processor will encrypt different data blocks using the same key. For example, in order to encrypt **n** number of data blocks with p processors, n/p data blocks will be encrypted by each processor. As each processor has its own data blocks and a key (increases data locality), all the 10/12/14 rounds (consists of four transformations) will be executed by each processor for encrypting each data block.

After encrypting all the data blocks of each processor, the encrypted data will be merged (Figure 5.4) in tree structure and return back to the main processor. For

example, if there are four processors working in parallel, processor P1 will send its encrypted data to P0 and P0 will merge its encrypted data with P1; processor P3 will send its encrypted data to P2, and P2 will merge its encrypted data with P3. Finally processor P2 will send its (P2 & P3) encrypted data to P0 and P0 will merge its (P0 & P1) encrypted data with P2. This technique of merging and returning data to the main processor will increase the concurrency and reduce the idle time of each processor.



a **b**

Figure 5.3 a) Data blocks are distributed between 2 processors
b) Data blocks are distributed among 4 processors

Decryption is done in the same way as the encryption. After decrypting the data blocks, the plain texts are merged and return back to the main processor in the same way as described above.

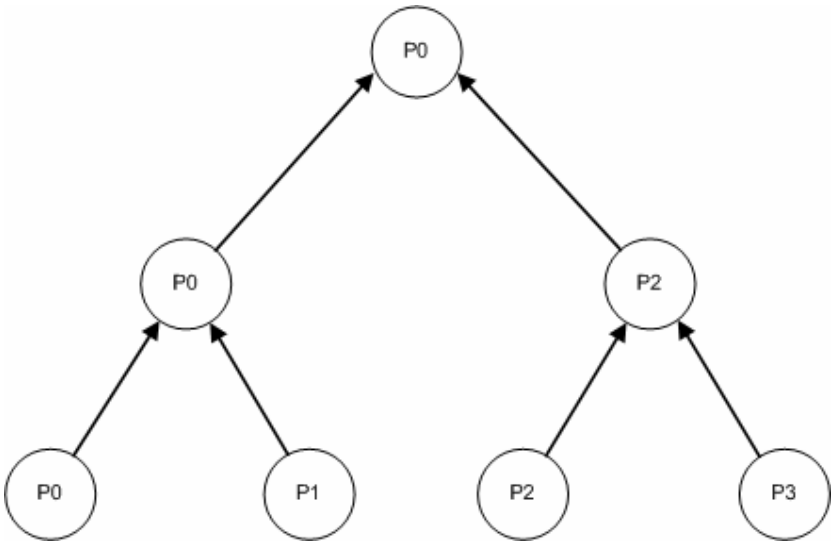


Figure 5.4 Encrypted data blocks are merged in tree structure

The overall parallel algorithm of AES cipher is described below:

Constant: **ArraySize = 160; int Nb = 4;**

int Nr = 10, **12**, or **14;** // rounds, for Nk = 4, 6, or 8

Inputs: **int nProcessors = 2/4/8/16 processors**

int tNumberOfBlocks // number of blocks to be encrypted

unsigned char key[16] // key for encrypting data

int k = 0;

array w of **4*Nb*(Nr+1)** bytes // expanded key

Internal work array: **my_pointer** is an array of pointers where each element of the array points to an array of data blocks. Each processor will have the variable **my_pointer**, where the first index will contain the data blocks for each processor.

Algorithm:

```
void Cipher(byte[] in, byte[] out, byte[] w) {
        nProcessors = 4
        int nBlockPerProcessor = tNumberOfBlocks / nProcessors
        int rank = processor's label
        if (rank = = processor 0){
                my_pointer[0] =  nBlockPerProcessor data blocks
                read the key
                send nBlockPerProcessor data blocks to rest of the processors
                send the same key to other processors
        }
       else{// for all other processors
                receive the nBlockPerProcessor data blocks from processor 0
                my_pointer[0] = nBlockPerProcessor data blocks
                receive the key from the processor 0
      }
      // each processor will execute this part of the algorithm.
        //Encrytion
        Encryption(my_pointer[0]);
        // Encrypted data are merged in tree structure and return back to the
        // main processor
         BTM(0, nProcessors -1);

        //Decryption
        Decryption(my_pointer[1]);
        BTM(0, nProcessors -1);}
```

```
void CreateStateMatrix(unsigned char block[],unsigned char
state[][4])
{       int count =0;
        for(int column=0; column<4; column++)
              for(int row=0; row<4;row++)
                    state[row][column] = block[count++] ;}
```

```
  void MergeEncryptedData(unsigned char Array[], unsigned char
                ArrayTwo[], unsigned char ArrayMain[],
                                          int FHIndex, int SHIndex){
        int numberOfData = 0;
        for(int count =0; count<= FHIndex;count++)
                ArrayMain[numberOfData++] = Array[count];

        for(count = 0; count<= SHIndex; count++)
                ArrayMain[numberOfData++] = ArrayTwo[count];
  }
```

```
  void BTM(int low, int high){
        int mid;
        if (low < high){
              mid = (low + high) / 2;
              BTM(low,mid);
              BTM(mid+1,high);
              if (my_rank == low){
                    my_pointer[++k] = new unsigned char[ArraySize * i];
                    MPI_Recv(my_pointer[k], ArraySize*i,
                              MPI_UNSIGNED_CHAR,mid+1,0,
                                        MPI_COMM_WORLD,&status);
                    my_pointer[++k] = new unsigned char[ArraySize * i* 2];
                    MergeEncryptedData(my_pointer[k-2],my_pointer[k-1],
                    my_pointer[k], ArraySize * i -1,ArraySize * i -1);
                    i *=2;
              }
              else if(my_rank == mid+1)
                      MPI_Send(my_pointer[k],ArraySize*i,
                              MPI_UNSIGNED_CHAR,low,0,MPI_COMM_WORLD);
      }
  }
```

```
void Encryption(unsigned char blocks[] ){
        int round = 1;
        unsigned char state[4][4]; // the state array
        int column;
        unsigned char block[16];

        //Creating the spaces for storing the encrypted data into the index no 1
        of    my_pointer
        my_pointer[++k] = new unsigned char[ArraySize];
        int toatlNoElement = 0;

        for(int row = 0; row<ArraySize;row +=16){
                for(column = 0; column<16; column++){
                        block[column] = blocks[row+column];
                }

                CreateStateMatrix(block,state);
                wCount = 0;
                AddRoundKey(state); // xor with expanded key
                for (; round < Nr; round++) {
                        SubBytesTransformation(state); // S-box substitution
                        ShiftRows(state); // mix up rows
                        MixColumns(state); // complicated mix of columns
                        AddRoundKey(state); // xor with expanded key
                }

                SubBytesTransformation(state); // S-box substitution
                ShiftRows(state); // mix up rows
                AddRoundKey(state);

                //store the encrypted data into the index no 1 of my_pointer
                for(int column= 0;column<Nb;column++)
                    for(int row = 0; row<Nb;row++);
```

```
void Decryption(unsigned char blocks[]){

        int round=1;
        unsigned char state[4][4]; // the state array
        int column;
        unsigned char block[16];

        //Creating the spaces for storing the plainText data into the index no
        //k+1 of my_pointer.
        my_pointer[++k] = new unsigned char[ArraySize];
        int toatlNoElement = 0;

        for(int row = 0; row<ArraySize;row +=16){
                for(column = 0; column<16; column++){
                        block[column] = blocks[row+column];
                }

                CreateStateMatrix(block,state);
                wCount = 4*Nb*(Nr+1); // count bytes during decryption
                InvAddRoundKey(state); // xor with expanded key
                for (round = Nr-1; round >= 1; round--) {
                        InvShiftRows(state); // mix up rows
                        InvSubBytes(state); // inverse S-box substitution
                        InvAddRoundKey(state); // xor with expanded key
                        InvMixColumns(state); // complicated mix of columns
                }
                InvShiftRows(state); // mix up rows
                InvSubBytes(state); // inverse S-box substitution
                InvAddRoundKey(state); // xor with expanded key

                //store the plaintext data into the index no k+1 of my_pointer
                for(int column= 0;column<Nb;column++)
                    for(int row = 0; row<Nb;row++);
                        my_pointer[k][toatlNoElement++] = state[row][column];
        }
}
```

5.4 Sample Input/Output

128-bit data, 128-bit Key
2 processors, each processor processes 4 data blocks
Encrypting . . .

Processor 0:

Plaintext1	**32**	**43**	**F6**	**A8**	**88**	**5A**	**30**	**8D**	**31**	**31**	**98**	**A2**	**E0**	**37**	**07**	**34**
Plaintext2	**38**	**21**	**1A**	**00**	**0B**	**23**	**DE**	**93**	**F7**	**B6**	**65**	**7D**	**F9**	**AE**	**C4**	**D1**
Plaintext3	**AF**	**DA**	**94**	**A5**	**E5**	**3C**	**A1**	**25**	**B0**	**39**	**D3**	**58**	**0**	**CE**	**BF**	**CA**
Plaintext4	**8E**	**9C**	**32**	**1E**	**84**	**47**	**CD**	**BC**	**9B**	**67**	**7E**	**B9**	**B6**	**23**	**5E**	**1A**
Key	**2B**	**7E**	**15**	**16**	**28**	**AE**	**D2**	**A6**	**AB**	**F7**	**15**	**88**	**09**	**CF**	**4F**	**3C**
Ciphertext1	**39**	**25**	**84**	**1D**	**02**	**DC**	**09**	**FB**	**DC**	**11**	**85**	**97**	**19**	**6A**	**0B**	**32**
Ciphertext2	**E3**	**97**	**D6**	**93**	**C8**	**82**	**E9**	**DF**	**8A**	**CD**	**3D**	**35**	**2A**	**20**	**0A**	**47**
Ciphertext3	**71**	**FB**	**5E**	**D7**	**3E**	**D0**	**76**	**AE**	**C5**	**A1**	**89**	**14**	**86**	**70**	**16**	**3F**
Ciphertext4	**DB**	**CA**	**D3**	**9F**	**C2**	**FC**	**B6**	**EF**	**F5**	**1B**	**60**	**39**	**53**	**1B**	**2B**	**24**

Processor 1:

Plaintext1	**D1**	**5B**	**5F**	**58**	**91**	**EA**	**82**	**C0**	**1F**	**28**	**4B**	**1A**	**37**	**B3**	**73**	**89**
Plaintext2	**3F**	**91**	**38**	**5A**	**E1**	**F3**	**7B**	**9C**	**3D**	**C2**	**AA**	**7B**	**9B**	**F2**	**7C**	**23**
Plaintext3	**7C**	**70**	**DC**	**B2**	**0F**	**CC**	**CA**	**20**	**5F**	**48**	**4F**	**E7**	**43**	**34**	**07**	**88**
Plaintext4	**46**	**BA**	**06**	**15**	**41**	**1F**	**0F**	**96**	**30**	**46**	**06**	**AA**	**3E**	**76**	**A8**	**72**
Key	**2B**	**7E**	**15**	**16**	**28**	**AE**	**D2**	**A6**	**AB**	**F7**	**15**	**88**	**09**	**CF**	**4F**	**3C**
Ciphertext1	**B4**	**FD**	**4B**	**E9**	**64**	**FF**	**4F**	**80**	**84**	**59**	**24**	**88**	**0F**	**21**	**54**	**F5**
Ciphertext2	**23**	**20**	**B7**	**96**	**CC**	**27**	**7A**	**91**	**E4**	**CC**	**1D**	**48**	**D4**	**75**	**3C**	**44**
Ciphertext3	**BC**	**BD**	**C4**	**72**	**EE**	**F1**	**A7**	**9F**	**51**	**FF**	**C3**	**2A**	**E7**	**B1**	**52**	**7C**
Ciphertext4	**0F**	**E9**	**FB**	**87**	**42**	**0F**	**AA**	**DD**	**0C**	**C6**	**9C**	**E1**	**40**	**F5**	**8B**	**E4**

Decrypting . . .
Processor 0:

Ciphertext1	**39**	**25**	**84**	**1D**	**02**	**DC**	**09**	**FB**	**DC**	**11**	**85**	**97**	**19**	**6A**	**0B**	**32**
Ciphertext2	**E3**	**97**	**D6**	**93**	**C8**	**82**	**E9**	**DF**	**8A**	**CD**	**3D**	**35**	**2A**	**20**	**0A**	**47**
Ciphertext3	**71**	**FB**	**5E**	**D7**	**3E**	**D0**	**76**	**AE**	**C5**	**A1**	**89**	**14**	**86**	**70**	**16**	**3F**
Ciphertext4	**DB**	**CA**	**D3**	**9F**	**C2**	**FC**	**B6**	**EF**	**F5**	**1B**	**60**	**39**	**53**	**1B**	**2B**	**24**
Key	**2B**	**7E**	**15**	**16**	**28**	**AE**	**D2**	**A6**	**AB**	**F7**	**15**	**88**	**09**	**CF**	**4F**	**3C**

Plaintext1	**32**	**43**	**F6**	**A8**	**88**	**5A**	**30**	**8D**	**31**	**31**	**98**	**A2**	**E0**	**37**	**07**	**34**
Plaintext2	**38**	**21**	**1A**	**00**	**0B**	**23**	**DE**	**93**	**F7**	**B6**	**65**	**7D**	**F9**	**AE**	**C4**	**D1**
Plaintext3	**AF**	**DA**	**94**	**A5**	**E5**	**3C**	**A1**	**25**	**B0**	**39**	**D3**	**58**	**00**	**CE**	**BF**	**CA**
Plaintext4	**8E**	**9C**	**32**	**1E**	**84**	**47**	**CD**	**BC**	**9B**	**67**	**7E**	**B9**	**B6**	**23**	**5E**	**1A**

Processor 1:

Ciphertext1	**B4**	**FD**	**4B**	**E9**	**64**	**FF**	**4F**	**80**	**84**	**59**	**24**	**88**	**0F**	**21**	**54**	**F5**
Ciphertext2	**23**	**20**	**B7**	**96**	**CC**	**27**	**7A**	**91**	**E4**	**CC**	**1D**	**48**	**D4**	**75**	**3C**	**44**
Ciphertext3	**BC**	**BD**	**C4**	**72**	**EE**	**F1**	**A7**	**9F**	**51**	**FF**	**C3**	**2A**	**E7**	**B1**	**52**	**7C**
Ciphertext4	**0F**	**E9**	**FB**	**87**	**42**	**0F**	**AA**	**DD**	**0C**	**C6**	**9C**	**E1**	**40**	**F5**	**8B**	**E4**
Key	**2B**	**7E**	**15**	**16**	**28**	**AE**	**D2**	**A6**	**AB**	**F7**	**15**	**88**	**09**	**CF**	**4F**	**3C**
Plaintext1	**D1**	**5B**	**5F**	**58**	**91**	**EA**	**82**	**C0**	**1F**	**28**	**4B**	**1A**	**37**	**B3**	**73**	**89**
Plaintext2	**3F**	**91**	**38**	**5A**	**E1**	**F3**	**7B**	**9C**	**3D**	**C2**	**AA**	**7B**	**9B**	**F2**	**7C**	**23**
Plaintext3	**7C**	**70**	**DC**	**B2**	**F**	**CC**	**CA**	**20**	**5F**	**48**	**4F**	**E7**	**43**	**34**	**7**	**88**
Plaintext4	**46**	**BA**	**6**	**15**	**41**	**1F**	**F**	**96**	**30**	**46**	**6**	**AA**	**3E**	**76**	**A8**	**72**

128-bit data, 192-bit Key
2 processors, each processor processes 4 data blocks
Encrypting . . .

Processor 0:

Plaintext1	**46**	**B7**	**15**	**DA**	**1A**	**1B**	**88**	**E4**	**15**	**23**	**C4**	**75**	**D7**	**C0**	**85**	**A5**
Plaintext2	**27**	**6B**	**F4**	**4E**	**8B**	**66**	**D6**	**6F**	**15**	**44**	**04**	**CE**	**7F**	**AB**	**1B**	**E2**
Plaintext3	**7B**	**59**	**74**	**EF**	**44**	**65**	**57**	**73**	**79**	**C7**	**6B**	**C1**	**2B**	**1B**	**71**	**EC**
Plaintext4	**35**	**A2**	**32**	**E6**	**CB**	**8A**	**B6**	**E9**	**56**	**6F**	**B7**	**16**	**7F**	**21**	**55**	**5D**
Key	**E9**	**55**	**A9**	**D5**	**0A**	**B2**	**91**	**60**	**E0**	**0B**	**3B**	**86**	**41**	**A8**	**5C**	**77**
	22	**5C**	**7A**	**AA**	**AE**	**54**	**FF**	**19**								
Ciphertext1	**FE**	**90**	**46**	**41**	**DB**	**FC**	**64**	**28**	**D7**	**52**	**97**	**48**	**25**	**77**	**C1**	**92**
Ciphertext2	**EF**	**AD**	**0A**	**1D**	**CA**	**8D**	**F1**	**97**	**CA**	**4C**	**26**	**E3**	**B0**	**9D**	**A3**	**D0**
Ciphertext3	**62**	**7B**	**8C**	**DE**	**B8**	**2D**	**32**	**28**	**52**	**11**	**A0**	**9A**	**EE**	**BE**	**AA**	**69**
Ciphertext4	**F1**	**6C**	**AC**	**BB**	**C6**	**30**	**7A**	**6E**	**BA**	**AD**	**A3**	**FA**	**C9**	**1A**	**24**	**11**

Processor 1:

Plaintext1	**8A**	**A8**	**C**	**9C**	**E6**	**85**	**E2**	**B**	**02**	**3B**	**E5**	**D7**	**62**	**0E**	**D4**	**0D**
Plaintext2	**F1**	**0B**	**1D**	**BA**	**99**	**48**	**07**	**51**	**11**	**6D**	**31**	**4C**	**F6**	**73**	**3B**	**16**
Plaintext3	**1E**	**AC**	**C3**	**29**	**29**	**4**	**93**	**73**	**58**	**86**	**18**	**FF**	**A3**	**22**	**17**	**D1**
Plaintext4	**05**	**AD**	**9B**	**12**	**1B**	**29**	**32**	**6D**	**ED**	**47**	**58**	**B8**	**0C**	**2A**	**34**	**D7**
Key	**E9**	**55**	**A9**	**D5**	**A**	**B2**	**91**	**60**	**E0**	**0B**	**3B**	**86**	**41**	**A8**	**5C**	**77**
	22	**5C**	**7A**	**AA**	**AE**	**54**	**FF**	**19**								
Ciphertext1	**EF**	**26**	**42**	**A5**	**62**	**DC**	**10**	**77**	**46**	**B0**	**14**	**13**	**A2**	**D2**	**69**	**86**
Ciphertext2	**C8**	**AC**	**E9**	**E3**	**6E**	**D8**	**9B**	**93**	**B9**	**B0**	**9D**	**63**	**05**	**F0**	**E6**	**14**
Ciphertext3	**8A**	**76**	**86**	**D3**	**E4**	**F2**	**85**	**D1**	**78**	**BA**	**52**	**0A**	**6C**	**6E**	**83**	**BE**
Ciphertext4	**32**	**CE**	**E2**	**54**	**8E**	**3A**	**F2**	**E6**	**13**	**66**	**A8**	**C3**	**FE**	**26**	**6B**	**6B**

Decrypting . . .
Processor 0:

Ciphertext1	**FE**	**90**	**46**	**41**	**DB**	**FC**	**64**	**28**	**D7**	**52**	**97**	**48**	**25**	**77**	**C1**	**92**
Ciphertext2	**EF**	**AD**	**0A**	**1D**	**CA**	**8D**	**F1**	**97**	**CA**	**4C**	**26**	**E3**	**B0**	**9D**	**A3**	**D0**
Ciphertext3	**62**	**7B**	**8C**	**DE**	**B8**	**2D**	**32**	**28**	**52**	**11**	**A0**	**9A**	**EE**	**BE**	**AA**	**69**
Ciphertext4	**F1**	**6C**	**AC**	**BB**	**C6**	**30**	**7A**	**6E**	**BA**	**AD**	**A3**	**FA**	**C9**	**1A**	**24**	**11**
Key	**E9**	**55**	**A9**	**D5**	**A**	**B2**	**91**	**60**	**E0**	**B**	**3B**	**86**	**41**	**A8**	**5C**	**77**
	22	**5C**	**7A**	**AA**	**AE**	**54**	**FF**	**19**								
Plaintext1	**46**	**B7**	**15**	**DA**	**1A**	**1B**	**88**	**E4**	**15**	**23**	**C4**	**75**	**D7**	**C0**	**85**	**A5**
Plaintext2	**27**	**6B**	**F4**	**4E**	**8B**	**66**	**D6**	**6F**	**15**	**44**	**4**	**CE**	**7F**	**AB**	**1B**	**E2**
Plaintext3	**7B**	**59**	**74**	**EF**	**44**	**65**	**57**	**73**	**79**	**C7**	**6B**	**C1**	**2B**	**1B**	**71**	**EC**
Plaintext4	**35**	**A2**	**32**	**E6**	**CB**	**8A**	**B6**	**E9**	**56**	**6F**	**B7**	**16**	**7F**	**21**	**55**	**5D**

Processor 1:

Ciphertext1	**EF**	**26**	**42**	**A5**	**62**	**DC**	**10**	**77**	**46**	**B0**	**14**	**13**	**A2**	**D2**	**69**	**86**
Ciphertext2	**C8**	**AC**	**E9**	**E3**	**6E**	**D8**	**9B**	**93**	**B9**	**B0**	**9D**	**63**	**05**	**F0**	**E6**	**14**
Ciphertext3	**8A**	**76**	**86**	**D3**	**E4**	**F2**	**85**	**D1**	**78**	**BA**	**52**	**0A**	**6C**	**6E**	**83**	**BE**
Ciphertext4	**32**	**CE**	**E2**	**54**	**8E**	**3A**	**F2**	**E6**	**13**	**66**	**A8**	**C3**	**FE**	**26**	**6B**	**6B**
Key	**E9**	**55**	**A9**	**D5**	**A**	**B2**	**91**	**60**	**E0**	**B**	**3B**	**86**	**41**	**A8**	**5C**	**77**
	22	**5C**	**7A**	**AA**	**AE**	**54**	**FF**	**19**								

Plaintext1	**8A**	**A8**	**C**	**9C**	**E6**	**85**	**E2**	**B**	**02**	**3B**	**E5**	**D7**	**62**	**0E**	**D4**	**0D**
Plaintext2	**F1**	**0B**	**1D**	**BA**	**99**	**48**	**07**	**51**	**11**	**6D**	**31**	**4C**	**F6**	**73**	**3B**	**16**
Plaintext3	**1E**	**AC**	**C3**	**29**	**29**	**4**	**93**	**73**	**58**	**86**	**18**	**FF**	**A3**	**22**	**17**	**D1**
Plaintext4	**05**	**AD**	**9B**	**12**	**1B**	**29**	**32**	**6D**	**ED**	**47**	**58**	**B8**	**0C**	**2A**	**34**	**D7**

128-bit data, 256-bit Key
2 processors, each processor processes 4 data blocks
Encrypting . . .

Processor 0:

Plaintext1	**DD**	**6E**	**82**	**DD**	**34**	**69**	**D1**	**75**	**24**	**B0**	**EE**	**81**	**16**	**DD**	**A0**	**00**
Plaintext2	**19**	**90**	**95**	**35**	**79**	**B4**	**9D**	**06**	**93**	**04**	**16**	**A2**	**E7**	**CD**	**A7**	**66**
Plaintext3	**6F**	**F5**	**32**	**02**	**30**	**3C**	**03**	**D7**	**0E**	**C2**	**4E**	**A4**	**E4**	**CA**	**D8**	**62**
Plaintext4	**D4**	**BD**	**F4**	**6C**	**DC**	**72**	**B1**	**E3**	**AA**	**AC**	**52**	**51**	**B2**	**E4**	**FE**	**8C**
Key	**7D**	**49**	**B9**	**DE**	**44**	**06**	**93**	**62**	**BD**	**C3**	**1F**	**B1**	**35**	**6E**	**27**	**BE**
	DF	**3B**	**9E**	**FF**	**6C**	**E9**	**D6**	**CE**	**DB**	**48**	**F4**	**E**	**93**	**F8**	**A0**	**11**
Ciphertext1	**08**	**79**	**D6**	**6E**	**FA**	**B4**	**61**	**71**	**BF**	**7E**	**A8**	**4A**	**A7**	**6B**	**67**	**FA**
Ciphertext2	**19**	**4E**	**5A**	**5C**	**3F**	**D6**	**3A**	**1F**	**05**	**B3**	**FD**	**AD**	**4E**	**95**	**1E**	**4E**
Ciphertext3	**55**	**7F**	**B9**	**AA**	**B6**	**C0**	**84**	**E8**	**E7**	**4C**	**AE**	**EA**	**46**	**6F**	**5C**	**51**
Ciphertext4	**EB**	**D7**	**25**	**9C**	**D1**	**B4**	**60**	**FF**	**95**	**3C**	**84**	**B5**	**56**	**E6**	**AE**	**B3**

Processor 1:

Plaintext1	**AF**	**73**	**01**	**B9**	**99**	**4D**	**E8**	**DE**	**DA**	**BB**	**4B**	**F0**	**30**	**53**	**F6**	**DE**
Plaintext2	**E1**	**12**	**7D**	**36**	**50**	**A3**	**75**	**8D**	**CE**	**DE**	**E3**	**21**	**B2**	**90**	**F5**	**BD**
Plaintext3	**AC**	**7A**	**F0**	**DD**	**56**	**9C**	**6A**	**13**	**0C**	**B3**	**B9**	**A9**	**FE**	**00**	**AB**	**88**
Plaintext4	**83**	**4B**	**78**	**58**	**B1**	**28**	**F4**	**E9**	**2A**	**79**	**09**	**D2**	**38**	**34**	**65**	**58**
Key	**7D**	**49**	**B9**	**DE**	**44**	**06**	**93**	**62**	**BD**	**C3**	**1F**	**B1**	**35**	**6E**	**27**	**BE**
	DF	**3B**	**9E**	**FF**	**6C**	**E9**	**D6**	**CE**	**DB**	**48**	**F4**	**E**	**93**	**F8**	**A0**	**11**
Ciphertext1	**C7**	**09**	**0B**	**F2**	**9A**	**C6**	**64**	**79**	**65**	**DD**	**E0**	**EE**	**D5**	**A1**	**0C**	**C7**
Ciphertext2	**9A**	**31**	**6C**	**B6**	**C3**	**C4**	**9D**	**46**	**46**	**5B**	**89**	**C2**	**D8**	**D1**	**92**	**93**
Ciphertext3	**8A**	**AB**	**C0**	**8C**	**37**	**AF**	**4D**	**A9**	**26**	**15**	**13**	**85**	**BC**	**C5**	**E9**	**BF**
Ciphertext4	**B1**	**80**	**5B**	**E7**	**7D**	**D6**	**D4**	**BD**	**BA**	**69**	**8D**	**D3**	**C6**	**5E**	**8F**	**7E**

Decrypting . . .
Processor 0:

Ciphertext1	**08**	**79**	**D6**	**6E**	**FA**	**B4**	**61**	**71**	**BF**	**7E**	**A8**	**4A**	**A7**	**6B**	**67**	**FA**
Ciphertext2	**19**	**4E**	**5A**	**5C**	**3F**	**D6**	**3A**	**1F**	**05**	**B3**	**FD**	**AD**	**4E**	**95**	**1E**	**4E**
Ciphertext3	**55**	**7F**	**B9**	**AA**	**B6**	**C0**	**84**	**E8**	**E7**	**4C**	**AE**	**EA**	**46**	**6F**	**5C**	**51**
Ciphertext4	**EB**	**D7**	**25**	**9C**	**D1**	**B4**	**60**	**FF**	**95**	**3C**	**84**	**B5**	**56**	**E6**	**AE**	**B3**
Key	**7D**	**49**	**B9**	**DE**	**44**	**06**	**93**	**62**	**BD**	**C3**	**1F**	**B1**	**35**	**6E**	**27**	**BE**
	DF	**3B**	**9E**	**FF**	**6C**	**E9**	**D6**	**CE**	**DB**	**48**	**F4**	**E**	**93**	**F8**	**A0**	**11**
Plaintext1	**DD**	**6E**	**82**	**DD**	**34**	**69**	**D1**	**75**	**24**	**B0**	**EE**	**81**	**16**	**DD**	**A0**	**00**
Plaintext2	**19**	**90**	**95**	**35**	**79**	**B4**	**9D**	**06**	**93**	**04**	**16**	**A2**	**E7**	**CD**	**A7**	**66**
Plaintext3	**6F**	**F5**	**32**	**02**	**30**	**3C**	**03**	**D7**	**0E**	**C2**	**4E**	**A4**	**E4**	**CA**	**D8**	**62**
Plaintext4	**D4**	**BD**	**F4**	**6C**	**DC**	**72**	**B1**	**E3**	**AA**	**AC**	**52**	**51**	**B2**	**E4**	**FE**	**8C**

Processor 1:

Ciphertext1	**C7**	**09**	**0B**	**F2**	**9A**	**C6**	**64**	**79**	**65**	**DD**	**E0**	**EE**	**D5**	**A1**	**0C**	**C7**
Ciphertext2	**9A**	**31**	**6C**	**B6**	**C3**	**C4**	**9D**	**46**	**46**	**5B**	**89**	**C2**	**D8**	**D1**	**92**	**93**
Ciphertext3	**8A**	**AB**	**C0**	**8C**	**37**	**AF**	**4D**	**A9**	**26**	**15**	**13**	**85**	**BC**	**C5**	**E9**	**BF**
Ciphertext4	**B1**	**80**	**5B**	**E7**	**7D**	**D6**	**D4**	**BD**	**BA**	**69**	**8D**	**D3**	**C6**	**5E**	**8F**	**7E**
Key	**7D**	**49**	**B9**	**DE**	**44**	**06**	**93**	**62**	**BD**	**C3**	**1F**	**B1**	**35**	**6E**	**27**	**BE**
	DF	**3B**	**9E**	**FF**	**6C**	**E9**	**D6**	**CE**	**DB**	**48**	**F4**	**E**	**93**	**F8**	**A0**	**11**
Plaintext1	**AF**	**73**	**01**	**B9**	**99**	**4D**	**E8**	**DE**	**DA**	**BB**	**4B**	**F0**	**30**	**53**	**F6**	**DE**
Plaintext2	**E1**	**12**	**7D**	**36**	**50**	**A3**	**75**	**8D**	**CE**	**DE**	**E3**	**21**	**B2**	**90**	**F5**	**BD**
Plaintext3	**AC**	**7A**	**F0**	**DD**	**56**	**9C**	**6A**	**13**	**0C**	**B3**	**B9**	**A9**	**FE**	**00**	**AB**	**88**
Plaintext4	**83**	**4B**	**78**	**58**	**B1**	**28**	**F4**	**E9**	**2A**	**79**	**09**	**D2**	**38**	**34**	**65**	**58**

128-bit data, 128-bit Key
4 processors, each processor processes 2 data blocks
Encrypting . . .

Processor 0:

Plaintext1	**1D**	**66**	**70**	**91**	**A4**	**78**	**81**	**C9**	**FC**	**B1**	**51**	**24**	**C6**	**FD**	**B0**	**85**
Plaintext2	**6D**	**ED**	**76**	**AF**	**B6**	**FE**	**BD**	**AA**	**98**	**1E**	**4D**	**69**	**4D**	**6C**	**58**	**A9**
Key	**83**	**AC**	**2F**	**9D**	**A2**	**19**	**CF**	**47**	**43**	**96**	**75**	**41**	**D2**	**B5**	**F8**	**55**
Ciphertext1	**F4**	**A3**	**3D**	**B4**	**43**	**8C**	**C9**	**6B**	**EB**	**3E**	**8B**	**A3**	**31**	**7D**	**7B**	**2C**
Ciphertext2	**A5**	**02**	**37**	**CC**	**79**	**6A**	**06**	**EE**	**B9**	**22**	**DE**	**AF**	**59**	**F9**	**53**	**49**

Processor 1:

Plaintext1	**A9**	**40**	**68**	**1C**	**AB**	**6B**	**95**	**C7**	**52**	**C0**	**FB**	**A8**	**B1**	**D3**	**6B**	**9D**
Plaintext2	**46**	**00**	**61**	**81**	**88**	**AF**	**36**	**98**	**BF**	**D7**	**97**	**2D**	**18**	**C1**	**36**	**79**
Key	**83**	**AC**	**2F**	**9D**	**A2**	**19**	**CF**	**47**	**43**	**96**	**75**	**41**	**D2**	**B5**	**F8**	**55**
Ciphertext1	**5B**	**FA**	**CC**	**49**	**76**	**68**	**EA**	**05**	**10**	**54**	**42**	**A9**	**ED**	**3D**	**4B**	**9A**
Ciphertext2	**2D**	**D7**	**EC**	**DA**	**DF**	**87**	**F3**	**9E**	**9E**	**11**	**4B**	**76**	**A6**	**1C**	**ED**	**83**

Processor 2:

Plaintext1	**F9**	**0F**	**BF**	**C7**	**92**	**FC**	**0E**	**D6**	**B5**	**E5**	**9F**	**7F**	**E7**	**09**	**46**	**97**
Plaintext2	**B7**	**8D**	**20**	**17**	**4E**	**C3**	**C9**	**7A**	**48**	**AB**	**CF**	**67**	**C2**	**BA**	**67**	**90**
Key	**83**	**AC**	**2F**	**9D**	**A2**	**19**	**CF**	**47**	**43**	**96**	**75**	**41**	**D2**	**B5**	**F8**	**55**
Ciphertext1	**A0**	**99**	**69**	**35**	**01**	**C5**	**18**	**7C**	**14**	**00**	**14**	**8F**	**20**	**D9**	**90**	**C6**
Ciphertext2	**A3**	**C4**	**AC**	**E5**	**18**	**E5**	**C4**	**34**	**8C**	**1A**	**FC**	**94**	**A3**	**F8**	**6E**	**D4**

Processor 3:

Plaintext1	**B4**	**DB**	**5F**	**DA**	**82**	**B4**	**54**	**BE**	**CD**	**2A**	**25**	**EF**	**8F**	**27**	**A7**	**3D**
Plaintext2	**67**	**9A**	**9B**	**B9**	**31**	**C0**	**DD**	**1A**	**DA**	**A3**	**DE**	**5F**	**72**	**E0**	**53**	**B6**
Key	**83**	**AC**	**2F**	**9D**	**A2**	**19**	**CF**	**47**	**43**	**96**	**75**	**41**	**D2**	**B5**	**F8**	**55**
Ciphertext1	**29**	**98**	**32**	**AC**	**98**	**29**	**E1**	**0D**	**2F**	**35**	**EB**	**A2**	**48**	**00**	**8F**	**C8**
Ciphertext2	**75**	**F1**	**B3**	**8F**	**BD**	**61**	**91**	**6F**	**D0**	**B4**	**D7**	**66**	**CE**	**AB**	**09**	**02**

Decrypting . . .
Processor 0:

Ciphertext1	**F4**	**A3**	**3D**	**B4**	**43**	**8C**	**C9**	**6B**	**EB**	**3E**	**8B**	**A3**	**31**	**7D**	**7B**	**2C**
Ciphertext2	**A5**	**02**	**37**	**CC**	**79**	**6A**	**06**	**EE**	**B9**	**22**	**DE**	**AF**	**59**	**F9**	**53**	**49**
Key	**83**	**AC**	**2F**	**9D**	**A2**	**19**	**CF**	**47**	**43**	**96**	**75**	**41**	**D2**	**B5**	**F8**	**55**
Plaintext1	**1D**	**66**	**70**	**91**	**A4**	**78**	**81**	**C9**	**FC**	**B1**	**51**	**24**	**C6**	**FD**	**B0**	**85**
Plaintext2	**6D**	**ED**	**76**	**AF**	**B6**	**FE**	**BD**	**AA**	**98**	**1E**	**4D**	**69**	**4D**	**6C**	**58**	**A9**

Processor 1:

Ciphertext1	**5B**	**FA**	**CC**	**49**	**76**	**68**	**EA**	**05**	**10**	**54**	**42**	**A9**	**ED**	**3D**	**4B**	**9A**
Ciphertext2	**2D**	**D7**	**EC**	**DA**	**DF**	**87**	**F3**	**9E**	**9E**	**11**	**4B**	**76**	**A6**	**1C**	**ED**	**83**
Key	**83**	**AC**	**2F**	**9D**	**A2**	**19**	**CF**	**47**	**43**	**96**	**75**	**41**	**D2**	**B5**	**F8**	**55**
Plaintext1	**1D**	**66**	**70**	**91**	**A4**	**78**	**81**	**C9**	**FC**	**B1**	**51**	**24**	**C6**	**FD**	**B0**	**85**
Plaintext2	**6D**	**ED**	**76**	**AF**	**B6**	**FE**	**BD**	**AA**	**98**	**1E**	**4D**	**69**	**4D**	**6C**	**58**	**A9**

Processor 2:

Ciphertext1	**A0**	**99**	**69**	**35**	**01**	**C5**	**18**	**7C**	**14**	**00**	**14**	**8F**	**20**	**D9**	**90**	**C6**
Ciphertext2	**A3**	**C4**	**AC**	**E5**	**18**	**E5**	**C4**	**34**	**8C**	**1A**	**FC**	**94**	**A3**	**F8**	**6E**	**D4**
Key	**83**	**AC**	**2F**	**9D**	**A2**	**19**	**CF**	**47**	**43**	**96**	**75**	**41**	**D2**	**B5**	**F8**	**55**
Plaintext1	**F9**	**0F**	**BF**	**C7**	**92**	**FC**	**0E**	**D6**	**B5**	**E5**	**9F**	**7F**	**E7**	**09**	**46**	**97**
Plaintext2	**B7**	**8D**	**20**	**17**	**4E**	**C3**	**C9**	**7A**	**48**	**AB**	**CF**	**67**	**C2**	**BA**	**67**	**90**

Processor 3:

Ciphertext1	**29**	**98**	**32**	**AC**	**98**	**29**	**E1**	**0D**	**2F**	**35**	**EB**	**A2**	**48**	**00**	**8F**	**C8**
Ciphertext2	**75**	**F1**	**B3**	**8F**	**BD**	**61**	**91**	**6F**	**D0**	**B4**	**D7**	**66**	**CE**	**AB**	**09**	**02**
Key	**83**	**AC**	**2F**	**9D**	**A2**	**19**	**CF**	**47**	**43**	**96**	**75**	**41**	**D2**	**B5**	**F8**	**55**
Plaintext1	**B4**	**DB**	**5F**	**DA**	**82**	**B4**	**54**	**BE**	**CD**	**2A**	**25**	**EF**	**8F**	**27**	**A7**	**3D**
Plaintext2	**67**	**9A**	**9B**	**B9**	**31**	**C0**	**DD**	**1A**	**DA**	**A3**	**DE**	**5F**	**72**	**E0**	**53**	**B6**

128-bit data, 192-bit Key
4 processors, each processor processes 2 data blocks
Encrypting . . .

Processor 0:

Plaintext1	**FF**	**32**	**BA**	**AF**	**59**	**30**	**D9**	**3E**	**1E**	**C5**	**AA**	**5D**	**51**	**B9**	**63**	**B3**
Plaintext2	**0E**	**8B**	**17**	**57**	**1B**	**B5**	**E8**	**38**	**BF**	**31**	**78**	**BD**	**D7**	**3C**	**A0**	**A8**
Key	**89**	**8B**	**BE**	**6B**	**C4**	**07**	**07**	**9F**	**3D**	**A1**	**67**	**2F**	**BF**	**CB**	**1F**	**A9**
	AC	**2C**	**C8**	**90**	**93**	**E5**	**8C**	**B7**								
Ciphertext1	**BB**	**B0**	**84**	**BB**	**20**	**8F**	**4D**	**49**	**C3**	**78**	**35**	**F1**	**C4**	**DC**	**42**	**C8**
Ciphertext2	**C3**	**FC**	**6B**	**C8**	**AA**	**54**	**7C**	**F4**	**AE**	**2D**	**EE**	**E5**	**57**	**50**	**54**	**C4**

Processor 1:

Plaintext1	**26**	**5A**	**65**	**40**	**ED**	**9A**	**27**	**AF**	**BB**	**1B**	**1C**	**A9**	**87**	**CF**	**FD**	**4E**
Plaintext2	**FC**	**7F**	**FF**	**52**	**14**	**10**	**05**	**5B**	**E8**	**4**	**14**	**AA**	**C7**	**44**	**7**	**FE**
Key	**89**	**8B**	**BE**	**6B**	**C4**	**07**	**07**	**9F**	**3D**	**A1**	**67**	**2F**	**BF**	**CB**	**1F**	**A9**
	AC	**2C**	**C8**	**90**	**93**	**E5**	**8C**	**B7**								
Ciphertext1	**E9**	**32**	**21**	**A4**	**81**	**54**	**08**	**8A**	**1F**	**97**	**2A**	**8B**	**8D**	**CE**	**D5**	**B1**
Ciphertext2	**EA**	**01**	**DE**	**61**	**09**	**73**	**7B**	**2E**	**86**	**EE**	**24**	**B5**	**DB**	**82**	**6E**	**4F**

Processor 2:

Plaintext1	**85**	**1C**	**84**	**B5**	**15**	**87**	**2E**	**35**	**5A**	**AD**	**1C**	**8A**	**3B**	**AA**	**8B**	**50**
Plaintext2	**F6**	**91**	**CF**	**D4**	**B6**	**B0**	**90**	**76**	**67**	**16**	**32**	**52**	**C9**	**53**	**95**	**1F**
Key	**89**	**8B**	**BE**	**6B**	**C4**	**07**	**07**	**9F**	**3D**	**A1**	**67**	**2F**	**BF**	**CB**	**1F**	**A9**
	AC	**2C**	**C8**	**90**	**93**	**E5**	**8C**	**B7**								
Ciphertext1	**0D**	**BD**	**35**	**1C**	**36**	**66**	**B2**	**47**	**75**	**CC**	**54**	**05**	**3A**	**40**	**1D**	**1C**
Ciphertext2	**9A**	**DA**	**9B**	**9F**	**8A**	**49**	**9E**	**22**	**CE**	**86**	**D4**	**CC**	**9F**	**34**	**A0**	**66**

Processor 3:

Plaintext1	**C3**	**7F**	**FF**	**56**	**FB**	**7E**	**56**	**97**	**A3**	**81**	**93**	**4A**	**96**	**D0**	**73**	**82**
Plaintext2	**A3**	**C8**	**6F**	**26**	**28**	**1F**	**EF**	**52**	**E3**	**70**	**BB**	**FB**	**06**	**F3**	**B2**	**D2**
Key	**89**	**8B**	**BE**	**6B**	**C4**	**07**	**07**	**9F**	**3D**	**A1**	**67**	**2F**	**BF**	**CB**	**1F**	**A9**
	AC	**2C**	**C8**	**90**	**93**	**E5**	**8C**	**B7**								
Ciphertext1	**CE**	**F4**	**9D**	**89**	**3E**	**B6**	**0A**	**C4**	**1E**	**3B**	**4C**	**1B**	**FB**	**75**	**D5**	**D8**
Ciphertext2	**C0**	**C3**	**F4**	**B4**	**34**	**5A**	**1F**	**DA**	**98**	**55**	**49**	**6E**	**7D**	**18**	**EB**	**47**

Decrypting . . .
Processor 0:

Ciphertext1	**BB**	**B0**	**84**	**BB**	**20**	**8F**	**4D**	**49**	**C3**	**78**	**35**	**F1**	**C4**	**DC**	**42**	**C8**
Ciphertext2	**C3**	**FC**	**6B**	**C8**	**AA**	**54**	**7C**	**F4**	**AE**	**2D**	**EE**	**E5**	**57**	**50**	**54**	**C4**
Key	**89**	**8B**	**BE**	**6B**	**C4**	**07**	**07**	**9F**	**3D**	**A1**	**67**	**2F**	**BF**	**CB**	**1F**	**A9**
	AC	**2C**	**C8**	**90**	**93**	**E5**	**8C**	**B7**								
Plaintext1	**FF**	**32**	**BA**	**AF**	**59**	**30**	**D9**	**3E**	**1E**	**C5**	**AA**	**5D**	**51**	**B9**	**63**	**B3**
Plaintext2	**0E**	**8B**	**17**	**57**	**1B**	**B5**	**E8**	**38**	**BF**	**31**	**78**	**BD**	**D7**	**3C**	**A0**	**A8**

Processor 1:

Ciphertext1	**E9**	**32**	**21**	**A4**	**81**	**54**	**08**	**8A**	**1F**	**97**	**2A**	**8B**	**8D**	**CE**	**D5**	**B1**
Ciphertext2	**EA**	**01**	**DE**	**61**	**09**	**73**	**7B**	**2E**	**86**	**EE**	**24**	**B5**	**DB**	**82**	**6E**	**4F**
Key	**89**	**8B**	**BE**	**6B**	**C4**	**07**	**07**	**9F**	**3D**	**A1**	**67**	**2F**	**BF**	**CB**	**1F**	**A9**
	AC	**2C**	**C8**	**90**	**93**	**E5**	**8C**	**B7**								
Plaintext1	**26**	**5A**	**65**	**40**	**ED**	**9A**	**27**	**AF**	**BB**	**1B**	**1C**	**A9**	**87**	**CF**	**FD**	**4E**
Plaintext2	**FC**	**7F**	**FF**	**52**	**14**	**10**	**05**	**5B**	**E8**	**4**	**14**	**AA**	**C7**	**44**	**7**	**FE**

Processor 2:

Ciphertext1	**0D**	**BD**	**35**	**1C**	**36**	**66**	**B2**	**47**	**75**	**CC**	**54**	**05**	**3A**	**40**	**1D**	**1C**
Ciphertext2	**9A**	**DA**	**9B**	**9F**	**8A**	**49**	**9E**	**22**	**CE**	**86**	**D4**	**CC**	**9F**	**34**	**A0**	**66**
Key	**89**	**8B**	**BE**	**6B**	**C4**	**07**	**07**	**9F**	**3D**	**A1**	**67**	**2F**	**BF**	**CB**	**1F**	**A9**
	AC	**2C**	**C8**	**90**	**93**	**E5**	**8C**	**B7**								
Plaintext1	**85**	**1C**	**84**	**B5**	**15**	**87**	**2E**	**35**	**5A**	**AD**	**1C**	**8A**	**3B**	**AA**	**8B**	**50**
Plaintext2	**F6**	**91**	**CF**	**D4**	**B6**	**B0**	**90**	**76**	**67**	**16**	**32**	**52**	**C9**	**53**	**95**	**1F**

Processor 3:

Ciphertext1	**CE**	**F4**	**9D**	**89**	**3E**	**B6**	**0A**	**C4**	**1E**	**3B**	**4C**	**1B**	**FB**	**75**	**D5**	**D8**
Ciphertext2	**C0**	**C3**	**F4**	**B4**	**34**	**5A**	**1F**	**DA**	**98**	**55**	**49**	**6E**	**7D**	**18**	**EB**	**47**
Key	**89**	**8B**	**BE**	**6B**	**C4**	**07**	**07**	**9F**	**3D**	**A1**	**67**	**2F**	**BF**	**CB**	**1F**	**A9**
	AC	**2C**	**C8**	**90**	**93**	**E5**	**8C**	**B7**								
Plaintext1	**C3**	**7F**	**FF**	**56**	**FB**	**7E**	**56**	**97**	**A3**	**81**	**93**	**4A**	**96**	**D0**	**73**	**82**
Plaintext2	**A3**	**C8**	**6F**	**26**	**28**	**1F**	**EF**	**52**	**E3**	**70**	**BB**	**FB**	**06**	**F3**	**B2**	**D2**

128-bit data, 256-bit Key
4 processors, each processor processes 2 data blocks
Encrypting . . .

Processor 0:

Plaintext1	**89**	**8B**	**BE**	**6B**	**C4**	**07**	**07**	**9F**	**3D**	**A1**	**67**	**2F**	**BF**	**CB**	**1F**	**A9**
Plaintext2	**AC**	**2C**	**C8**	**90**	**93**	**E5**	**8C**	**B7**	**05**	**18**	**95**	**EE**	**A5**	**AB**	**C6**	**E0**
Key	**99**	**F9**	**A4**	**31**	**D**	**60**	**61**	**62**	**85**	**15**	**29**	**91**	**A2**	**6**	**C3**	**BA**
	7B	**B8**	**31**	**2C**	**38**	**8B**	**E0**	**EA**	**DD**	**3E**	**21**	**52**	**63**	**B5**	**DD**	**FC**
Ciphertext1	**98**	**31**	**6D**	**EB**	**23**	**05**	**5B**	**A9**	**AC**	**4C**	**16**	**EC**	**6A**	**E7**	**FA**	**75**
Ciphertext2	**32**	**F6**	**91**	**F2**	**EE**	**49**	**E3**	**7F**	**38**	**1A**	**F2**	**4C**	**00**	**64**	**AB**	**0A**

Processor 1:

Plaintext1	**7F**	**49**	**A9**	**3C**	**9B**	**AA**	**A9**	**13**	**D1**	**15**	**82**	**82**	**DE**	**23**	**F5**	**90**
Plaintext2	**B8**	**C5**	**C0**	**59**	**1F**	**88**	**CD**	**6D**	**75**	**18**	**AA**	**73**	**CF**	**03**	**39**	**12**
Key	**99**	**F9**	**A4**	**31**	**D**	**60**	**61**	**62**	**85**	**15**	**29**	**91**	**A2**	**6**	**C3**	**BA**
	7B	**B8**	**31**	**2C**	**38**	**8B**	**E0**	**EA**	**DD**	**3E**	**21**	**52**	**63**	**B5**	**DD**	**FC**
Ciphertext1	**EA**	**4D**	**6A**	**10**	**4C**	**2A**	**53**	**6A**	**2F**	**F7**	**27**	**B4**	**4B**	**10**	**31**	**31**
Ciphertext2	**F0**	**18**	**02**	**AD**	**99**	**FA**	**3C**	**70**	**A0**	**C2**	**66**	**06**	**8A**	**C6**	**D0**	**30**

Processor 2:

Plaintext1	**4C**	**C0**	**A9**	**11**	**A6**	**F0**	**A4**	**BC**	**6**	**E3**	**CA**	**8A**	**1D**	**5E**	**5F**	**FE**
Plaintext2	**70**	**9C**	**40**	**CB**	**F3**	**92**	**1C**	**3B**	**DA**	**78**	**E0**	**D1**	**AC**	**84**	**73**	**2E**
Key	**99**	**F9**	**A4**	**31**	**D**	**60**	**61**	**62**	**85**	**15**	**29**	**91**	**A2**	**6**	**C3**	**BA**
	7B	**B8**	**31**	**2C**	**38**	**8B**	**E0**	**EA**	**DD**	**3E**	**21**	**52**	**63**	**B5**	**DD**	**FC**
Ciphertext1	**4C**	**E5**	**41**	**1B**	**61**	**EA**	**BB**	**EE**	**39**	**22**	**5C**	**61**	**84**	**8A**	**55**	**51**
Ciphertext2	**50**	**D3**	**98**	**8D**	**C5**	**42**	**41**	**74**	**AC**	**AB**	**F4**	**F2**	**D7**	**54**	**50**	**E1**

Processor 3:

Plaintext1	**0D**	**FA**	**83**	**E5**	**F9**	**83**	**46**	**CD**	**F6**	**B3**	**86**	**DD**	**95**	**A2**	**8B**	**8D**
Plaintext2	**83**	**21**	**79**	**C5**	**15**	**F9**	**80**	**42**	**25**	**74**	**54**	**3C**	**1D**	**1F**	**DD**	**46**
Key	**99**	**F9**	**A4**	**31**	**D**	**60**	**61**	**62**	**85**	**15**	**29**	**91**	**A2**	**6**	**C3**	**BA**
	7B	**B8**	**31**	**2C**	**38**	**8B**	**E0**	**EA**	**DD**	**3E**	**21**	**52**	**63**	**B5**	**DD**	**FC**
Ciphertext1	**5E**	**C4**	**AD**	**74**	**4A**	**17**	**68**	**B7**	**20**	**EE**	**A2**	**FA**	**B1**	**A2**	**55**	**35**
Ciphertext2	**63**	**67**	**7A**	**56**	**97**	**91**	**C7**	**A9**	**E2**	**AB**	**70**	**23**	**1D**	**0C**	**D2**	**E7**

Decrypting . . .
Processor 0:

Ciphertext1	**98**	**31**	**6D**	**EB**	**23**	**05**	**5B**	**A9**	**AC**	**4C**	**16**	**EC**	**6A**	**E7**	**FA**	**75**
Ciphertext2	**32**	**F6**	**91**	**F2**	**EE**	**49**	**E3**	**7F**	**38**	**1A**	**F2**	**4C**	**00**	**64**	**AB**	**0A**
Key	**99**	**F9**	**A4**	**31**	**D**	**60**	**61**	**62**	**85**	**15**	**29**	**91**	**A2**	**6**	**C3**	**BA**
	7B	**B8**	**31**	**2C**	**38**	**8B**	**E0**	**EA**	**DD**	**3E**	**21**	**52**	**63**	**B5**	**DD**	**FC**
Plaintext1	**89**	**8B**	**BE**	**6B**	**C4**	**7**	**7**	**9F**	**3D**	**A1**	**67**	**2F**	**BF**	**CB**	**1F**	**A9**
Plaintext2	**AC**	**2C**	**C8**	**90**	**93**	**E5**	**8C**	**B7**	**5**	**18**	**95**	**EE**	**A5**	**AB**	**C6**	**E0**

Processor 1:

Ciphertext1	**EA**	**4D**	**6A**	**10**	**4C**	**2A**	**53**	**6A**	**2F**	**F7**	**27**	**B4**	**4B**	**10**	**31**	**31**
Ciphertext2	**F0**	**18**	**02**	**AD**	**99**	**FA**	**3C**	**70**	**A0**	**C2**	**66**	**06**	**8A**	**C6**	**D0**	**30**
Key	**99**	**F9**	**A4**	**31**	**D**	**60**	**61**	**62**	**85**	**15**	**29**	**91**	**A2**	**6**	**C3**	**BA**
	7B	**B8**	**31**	**2C**	**38**	**8B**	**E0**	**EA**	**DD**	**3E**	**21**	**52**	**63**	**B5**	**DD**	**FC**
Plaintext1	**7F**	**49**	**A9**	**3C**	**9B**	**AA**	**A9**	**13**	**D1**	**15**	**82**	**82**	**DE**	**23**	**F5**	**90**
Plaintext2	**B8**	**C5**	**C0**	**59**	**1F**	**88**	**CD**	**6D**	**75**	**18**	**AA**	**73**	**CF**	**3**	**39**	**12**

Processor 2:

Ciphertext1	**4C**	**E5**	**41**	**1B**	**61**	**EA**	**BB**	**EE**	**39**	**22**	**5C**	**61**	**84**	**8A**	**55**	**51**
Ciphertext2	**50**	**D3**	**98**	**8D**	**C5**	**42**	**41**	**74**	**AC**	**AB**	**F4**	**F2**	**D7**	**54**	**50**	**E1**
Key	**99**	**F9**	**A4**	**31**	**D**	**60**	**61**	**62**	**85**	**15**	**29**	**91**	**A2**	**6**	**C3**	**BA**
	7B	**B8**	**31**	**2C**	**38**	**8B**	**E0**	**EA**	**DD**	**3E**	**21**	**52**	**63**	**B5**	**DD**	**FC**
Plaintext1	**4C**	**C0**	**A9**	**11**	**A6**	**F0**	**A4**	**BC**	**6**	**E3**	**CA**	**8A**	**1D**	**5E**	**5F**	**FE**
Plaintext2	**70**	**9C**	**40**	**CB**	**F3**	**92**	**1C**	**3B**	**DA**	**78**	**E0**	**D1**	**AC**	**84**	**73**	**2E**

Processor 3:

Ciphertext1	**5E**	**C4**	**AD**	**74**	**4A**	**17**	**68**	**B7**	**20**	**EE**	**A2**	**FA**	**B1**	**A2**	**55**	**35**
Ciphertext2	**63**	**67**	**7A**	**56**	**97**	**91**	**C7**	**A9**	**E2**	**AB**	**70**	**23**	**1D**	**0C**	**D2**	**E7**
Key	**99**	**F9**	**A4**	**31**	**D**	**60**	**61**	**62**	**85**	**15**	**29**	**91**	**A2**	**6**	**C3**	**BA**
	7B	**B8**	**31**	**2C**	**38**	**8B**	**E0**	**EA**	**DD**	**3E**	**21**	**52**	**63**	**B5**	**DD**	**FC**
Plaintext1	**0D**	**FA**	**83**	**E5**	**F9**	**83**	**46**	**CD**	**F6**	**B3**	**86**	**DD**	**95**	**A2**	**8B**	**8D**
Plaintext2	**83**	**21**	**79**	**C5**	**15**	**F9**	**80**	**42**	**25**	**74**	**54**	**3C**	**1D**	**1F**	**DD**	**46**

5.5 Run Time Complexity of the Parallel Implementation

Time complexity is the most important measure of the performance of a parallel algorithm, since the primary motivation for parallel computation is to achieve a speedup in the computation. Parallel algorithms are executed by a set of processors and usually require inter-processor data transfers to complete execution successfully. The time complexity of a parallel algorithm to solve a problem of size n is a function T(n,p) which is the maximum time that elapses between the start of the algorithm's execution by one processor and its termination by one or more processors with regard to any arbitrary input. There are two different kinds of operations associated with parallel algorithms:

- Elementary operation
- Data routing operation

Elementary operation is an arithmetic or logical operation performed locally by a processor. **Data routing operations** refer to the routing of data among processors for exchanging the information. The time complexity of a parallel algorithm is determined by counting both elementary steps and data routing steps. A corollary follows that the time complexity of a parallel algorithm depends on the type of computational model being used as well as on the number of processors available.

Parallel computations are usually structured as a set of tasks executing concurrently and cooperatively on concurrent systems. Besides the actual service time spent in the system resources, execution time of a parallel computation consists of two kinds of additional delay. Queuing delay results when two or more tasks compete for resources in the system. Synchronization results when a task has to idle and wait for others to finish before continuing. Because of the presence of queuing and synchronization delays, execution times of parallel computations are very difficult to predict.

It is not possible to measure the performance of a parallel algorithm just by evaluating the run time complexity. It is also important to evaluate the speedup factor and efficiency of the algorithm. All of them are described below:

Run Time

The serial run time of a program is the time elapsed between the beginning and the end of its execution on a sequential computer. The parallel run time is the time that elapses from the moment that a parallel computation starts to the moment that the last processor finishes execution. The serial and parallel run time is denoted by T_S and T_P respectively.

Speed-up

When evaluating a parallel system, it is often important to know how much performance gain is achieved by parallelizing a given application over a sequential implementation. Speed-up is defined as the ratio of the time taken to solve a problem on a single processor to the time required to solve the same problem on a parallel computer with p identical processors. The speedup is denoted by the symbol S. Therefore,

$$S = T_S / T_P$$

Formally, the speedup S is defined as the ratio of the serial run time of the **best sequential algorithm** for solving a problem to the time taken by the parallel algorithm to solve the same problem on P processors.

Efficiency

Efficiency is defined as the Speed-up with N processors divided by the number of processors N. Conceptually, the efficiency of the algorithm measures how well all N processors are being used when the algorithm is computed in parallel. An efficiency of 100 percent means that all of the processors are being fully used all the time. Efficiency is denoted by E. Therefore,

$$E = S / P$$

$$= T_S / P T_P$$

In an ideal parallel system, speedup is equal to p and efficiency is equal to 1. In practice, ideal behavior is not achieved because while executing a parallel algorithm, the processors cannot devote 100 percent of their time to the computations of the algorithm. Because, some part of the time is spent in inter-processor communication.

Parallel run time of AES:

Parallel run time of the AES algorithm is the sum of the time to distribute the data blocks among the processors, time to encrypt the data blocks and the time to merge them in tree structure and returning the data blocks to the main processor. The time taken to communicate a message between two processors in the network is called the communication latency which consists of startup time (t_s), per-hop time (t_h) and per-word transfer time (t_w).

The **startup time** is the time required to handle a message at the sending processor. This includes the time to prepare the message (adding header, trailer, and error correction information), the time to execute the routing algorithm, and the time to establish an interface between the local processor and the router. This delay is incurred only once for a single message transfer. After a message leaves a processor, ity takes a finite amount of time to reach the next processor inits path. The time taken by the header of a message to travel between two directly-connected processors in the network is called the **per-hop time**. If the channel bandwidth is r words per second, then each word takes t = 1/r to traverse the link. This time is called **per-word transfer time**. Therefore, the parallel run time of AES consists of communication time, encryption time and the merge time:

$$T_p = T_{comm} + T_{encrypt} + T_{merge}$$

Assume that

N data blocks are encrypted.

P processors are working in parallel.

For distributing N data blocks among the P processors, N/P data blocks are sent to each processor. P-1 send operations are performed because the main processor is

already containing its own data. Each data block consists of 4 words. Hence the communication time is

$$T_{comm.} = (t_s + t_h + 4t_wN/P)(P-1)$$

For encrypting one data block, except the AddRoundKey transformation, all the transformations are executed 10 times. AddRoundkey transformation is executed 11 times:

$$\text{Number of operations} = 10*16*N + 11 * 16 * N + (10 * 8 * 4)N + 10 * 4 * (20 + 4)N$$

$$= 1616N$$

Therefore, the total encryption time for N/P data blocks is:

$$T_{encrypt} = N/P\ (1616N)$$

Encrypted data blocks are merged and return back to the main processor in tree structure. Therefore the merge time is:

$$T_{Merge} = N/P \log N/P + (P-1)\ (t_s + t_h + 4t_wN/p)$$

So,

$$T_p = (t_s + t_h + 4t_wN/P)(P-1) + N/P\ (1616N) + N/P \log N/P + (P-1)\ (t_s + t_h + 4t_wN/p)$$

Serial run time of AES algorithm consists of the encryption time only. Hence the serial run time is:

$$T_s = 1616N$$

Speedup $S = T_s / T_p$

$$= (1616N) /((t_s + t_h + 4t_wN/P)(P-1) + N/P\ (1616N) + N/P \log N/P + (P-1)\ (t_s + t_h + 4t_wN/p))$$

Efficiency $E = (1616N) / P * ((t_s + t_h + 4t_wN/P)(P-1) + N/P\ (1616N) + N/P \log N/P + (P-1)\ (t_s + t_h + 4t_wN/p))$

Performance of parallel implementation of AES:

The AES algorithm is implemented in a parallel machine with the following configuration:

Hardware Configuration

Processor : Pentium III **Clock Speed** : 800 MHz

Ram : 128 MB **Cache** : 256 KB

Parallel Architecture : MPI Based SIMD Cluster

Interconnection Network : Star

Table 5.1 Performance of AES in a parallel machine with 2 processors

Key Size	**Number of Data Blocks**						
128 bit	**5000**	**10000**	**15000**	**20000**	**25000**	**30000**	**50000**
	ms	**ms**	**ms**	**ms**	**ms**	**ms**	**ms**
Test1	283	578	873	1132	1406	1743	2851
Test2	279	580	882	1140	1402	1734	2859
Test3	275	578	880	1130	1406	1750	2867
Avg.	279	579	878	1134	1405	1742	2859
Time/ Block	55.8 μs	57.9 μs	58.5 μs	56.7 μs	56.2 μs	58.0 μs	57.1 μs
Speedup	1.64	1.57	1.54	1.58	1.60	1.55	1.58
Efficiency	0.82	0.79	0.77	0.79	0.80	0.78	0.79
192 bit	**5000**	**10000**	**15000**	**20000**	**25000**	**30000**	**50000**
	ms	**ms**	**ms**	**ms**	**ms**	**ms**	**ms**
Test1	360	734	1035	1424	1716	2099	3587
Test2	350	729	1030	1424	1723	2089	3582
Test3	355	731	1040	1424	1716	2096	3574
Avg.	355	731	1035	1424	1718	2095	3581
Time/ Block	71.0 μs	73.1 μs	69.0 μs	71.2 μs	68.7 μs	69.8 μs	71.6 μs
Speedup	1.57	1.54	1.59	1.55	1.60	1.57	1.54
Efficiency	0.78	0.77	0.79	0.78	0.80	0.79	0.77
256 bit	**5000**	**10000**	**15000**	**20000**	**25000**	**30000**	**50000**
	ms	**ms**	**ms**	**ms**	**ms**	**ms**	**ms**
Test1	425	827	1245	1620	2063	2518	4074
Test2	425	817	1235	1625	2063	2518	4079
Test3	430	823	1257	1621	2075	2518	4082
Avg.	427	822	1246	1622	2067	2518	4078
Time/ Block	85.4 μs	82.2 μs	83.0 μs	81.1 μs	82.6 μs	83.9 μs	81.5 μs
Speedup	1.54	1.58	1.56	1.60	1.56	1.54	1.59

Efficiency	0.77	0.79	0.78	0.80	0.78	0.77	0.79

Table 5.2 Performance of AES in a parallel machine with 4 processors

Key Size	Number of Data Blocks						
128 bit	5000	10000	15000	20000	25000	30000	50000
	ms	ms	ms	ms	ms	ms	ms
Test1	146	285	443	581	728	891	1470
Test2	142	296	450	576	731	899	1463
Test3	150	293	440	584	731	900	1473
Avg.	146	291	444	580	730	897	1469
Time/ Block	29.2 μs	29.1 μs	29.6 μs	29.0 μs	29.2 μs	29.9 μs	29.4 μs
Speedup	3.14	3.13	3.05	3.10	3.08	3.01	3.07
Efficiency	0.78	0.78	0.76	0.77	0.77	0.75	0.77
192 bit	5000	10000	15000	20000	25000	30000	50000
	ms	ms	ms	ms	ms	ms	ms
Test1	179	374	529	733	917	1061	1820
Test2	190	367	527	720	906	1067	1800
Test3	187	380	540	723	915	1070	1805
Avg.	185	374	532	725	913	1066	1808
Time/ Block	37.1 μs	37.4 μs	35.5 μs	36.3 μs	36.5 μs	35.5 μs	36.2 μs
Speedup	3.01	3.01	3.09	3.04	3.01	3.09	3.04
Efficiency	0.75	0.75	0.77	0.76	0.75	0.77	0.76
256 bit	5000	10000	15000	20000	25000	30000	50000
	ms	ms	ms	ms	ms	ms	ms
Test1	210	421	650	836	1055	1278	2093
Test2	212	430	653	845	1065	1288	2085
Test3	220	418	642	835	1067	1275	2095
Avg.	214	423	648	839	1062	1280	2091
Time/ Block	42.8 μs	42.3 μs	43.2 μs	41.9 μs	42.5 μs	42.7 μs	41.8 μs
Speedup	3.07	3.07	3.00	3.09	3.03	3.03	3.09
Efficiency	0.77	0.77	0.75	0.77	0.76	0.76	0.77

5.6 Concluding Remarks

From the above two tables, it is found that the efficiency of parallel AES ranges from 0.75 to 0.82 and the speedup is less than the number of processors used in the parallel implementation. In practice, it is not possible to achieve speedup equal to P and efficiency equal to 1, because while executing a parallel algorithm, the processors cannot devote 100 percent of their time to the computations of the algorithm. The time to transfer data between processors is equally the most significant source of parallel processing overhead. Another reason is load unbalancing i.e. not all the processors are equally busy while executing a parallel program. If different processors have different work loads, some processors may be idle during part of the time that others are working on the problem. It is very difficult to develop any parallel algorithm where each processor will work equal amount of time.

CHAPTER 6

CONCLUSION

6.1 Summery

The symmetric-key cryptography is efficient for encryption while the Public-key cryptography facilitates efficient signatures (particularly non-repudiation) and key management. On the other hand, the most widely used symmetric-key encryption technique like DES is vulnerable to a brute-force attack because of its inadequate key size compares to the processing power of modern computers. The mainstream cryptographic community has long held that DES's 56-bit key is too short to withstand a brute-force attack from modern computers. In mid 1990, RSA Laboratories of USA sponsored a series of cryptographic challenges to prove that DES was no longer appropriate for use.

In order to increase the security of symmetric-key cryptography, National Institute of Standards and Technology (NIST) of USA in 1997 issued a call for proposals for a new Advanced Encryption Standard (AES), which should have security strength better than DES and significantly improved efficiency. **AES** is a block cipher adopted as an encryption standard by the US government, and is expected to be used worldwide and analysed extensively, as was the case with its predecessor, the Data Encryption Standard (DES). AES has a fixed block size of 128 bits and a key size of 128, 192 or 256 bits. Unlike its predecessor DES, it is not a Feistel network, but a substitution-permutation network. AES is fast in both software and hardware, is relatively easy to implement, and requires little memory. As a new encryption standard, it is currently being deployed on a large scale.

As of October 2002, there are no known successful attacks against AES. NSA (National Security Agency) of USA reviewed all the AES finalists, including Rijndael, and stated that all of them are secure enough for US Government non-classified

data. In June 2003, the US Government announced [14] that AES may be used for classified information.

After implementing the AES algorithm on a single processor, it is found that the AES algorithm has a linear complexity that means when the value of N (number of data blocks) ranges from 10 – 100000, the execution time will vary from 10^{-5} second to 1 seconds (each operation is assumed to take 10^{-6} second). However, when the value of N is greater than 10^{8}, the execution time of the algorithm will require several days to encrypt or decrypt. This creates the reason for implementing the algorithm in parallel.

While implementing the AES algorithm in parallel, it is found that it is not possible to achieve speedup equal to P (number of processors) and efficiency equal to 1, because while executing a parallel algorithm, the processors cannot devote 100 percent of their time to the computations of the algorithm. The time to transfer data between processors is equally the most significant source of parallel processing overhead. Another reason is load unbalancing i.e. not all the processors are equally busy while executing a parallel program. If different processors have different workloads, some processors may be idle during part of the time that others are working on the problem. It is very difficult to develop any parallel algorithm where each processor will work equal amount of time.

6.2 Conclusion

After implementing the AES algorithm in parallel, it is found that the performance of AES algorithm increases significantly as the number of processor increases. It is not possible to get the speedup factor equal to P (number of processor), as some parallel processing overhead is also occurred during the implementation of AES in parallel. In figure 6.1 the performance of serial implementation of AES is shown. In figure 6.2 and 6.3, the performance of parallel implementation of AES is shown with 2 and 4

processors. The speedup factor of AES is given in figure 6.4 and 6.5 with 2 and 4 processors.

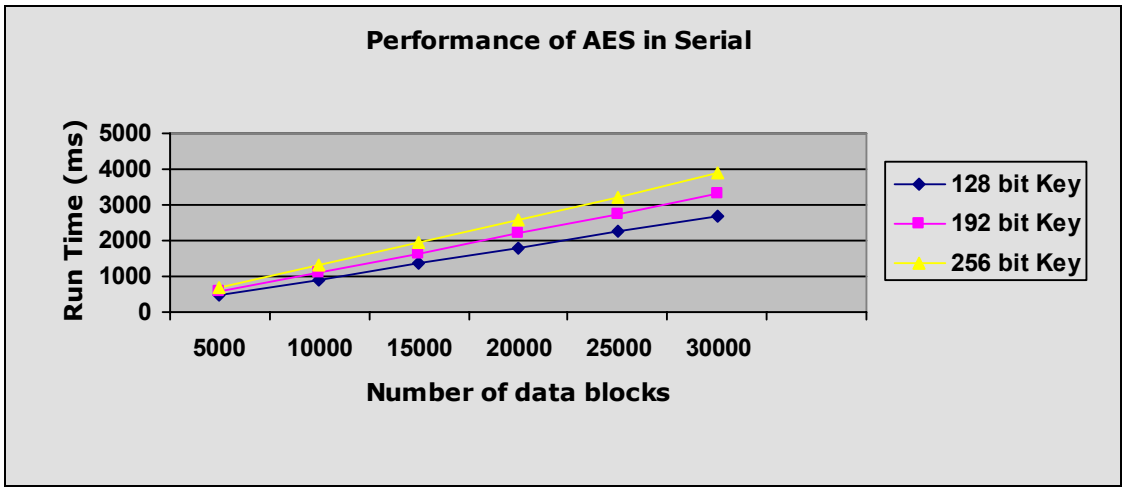


Figure 6.1 Performance of AES in Serial

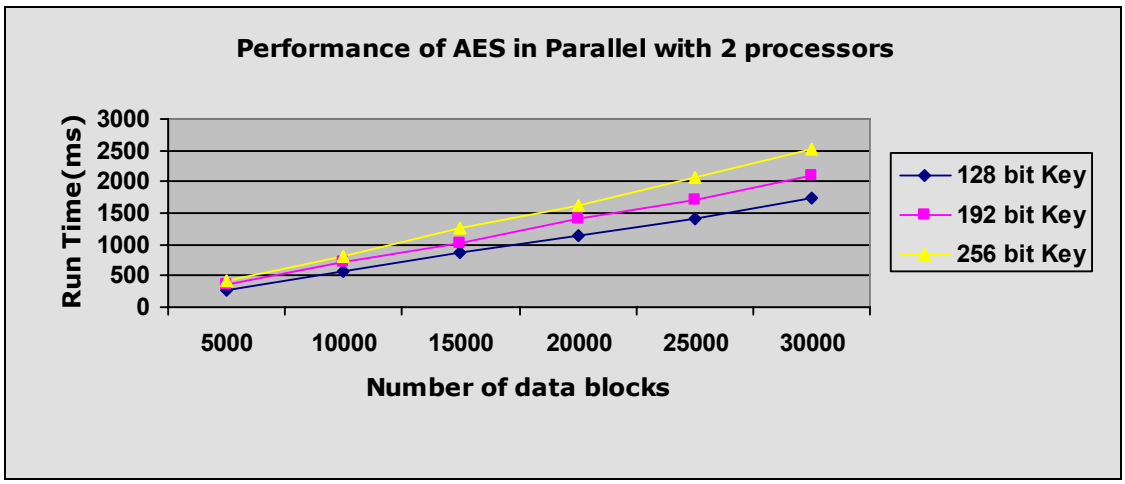


Figure 6.2 Performance of AES in parallel with 2 processors

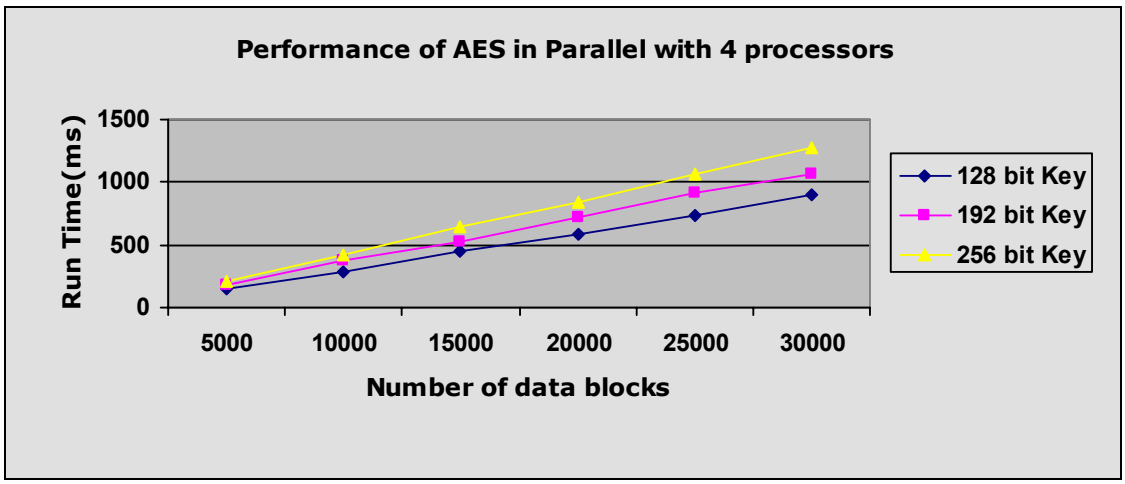


Figure 6.3 Performance of AES in parallel with 4 processors

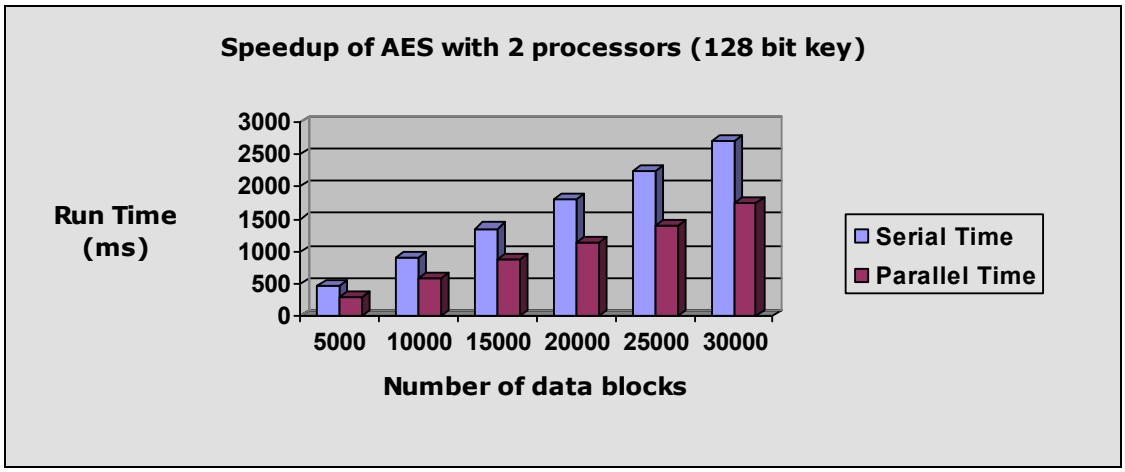


Figure 6.4 Speedup of AES with 2 processors

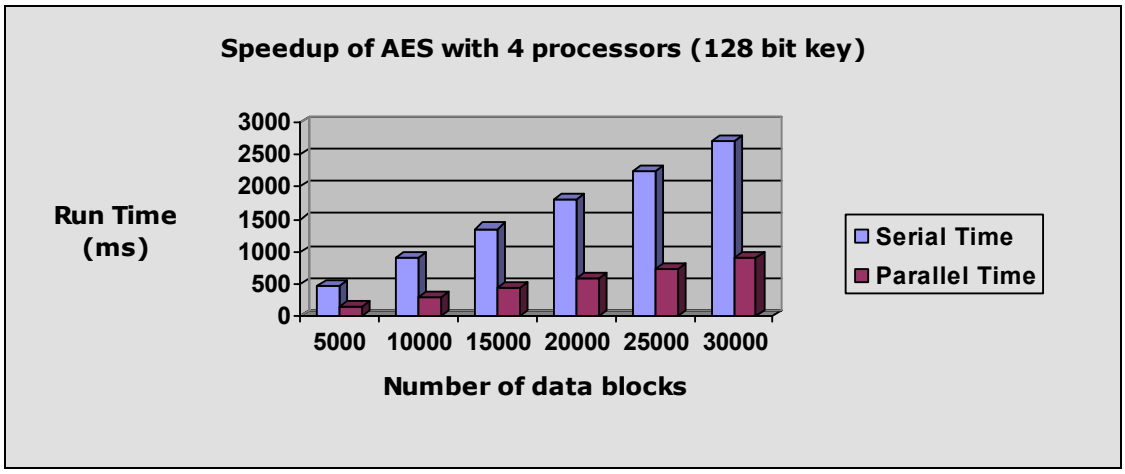


Figure 6.5 Speedup of AES with 4 processors

6.3 Suggestion for Future Work

As the performance of any parallel implementation depends on the parallel architectures and the interconnection networks, the same parallel implementation can give different performance on different architectures and interconnection networks. So, there is a good opportunity to work on some other architectures and interconnection networks and find out which architecture and interconnection network will be suitable for parallel implementation of AES.

REFERENCES

1. Menezes, A. and Vanstone, S. "Handbook of Applied Cryptography", CRC Press, Inc. 1996

2. National Bureau of Standards, NBS FIPS PUB 46, "Data Encryption Standard,", U.S. Department of Commerce, January 1977.

3. Coppersmith, D. "The Data Encryption Standard (DES) and Its Strength Against Attacks." IBM Journal of Research and Development, May 1994.

4. Diffie, W. and Hellman, M. "Multiuser Cryptographic Techniques" proceedings of AFIPS National Computer Conference, 1976, 109-112

5. Korner. T. "The Pleasures of Counting", Cambridge, England, Cambridge University Press, 1996

6. Khan, D. "The Codebreakers: The Story of Secret Writing." New York, 1996

7. Schneier, B. "Applied Cryptgraphy." New York: Wiley, 1996.

8. Stallings, W. "Cryptography and Network Security: Principles and Practices." Third Edition, Pearson Education, Inc. 2003.

9. Rivest, R., Shamir, A. and Adleman, L. "A Method for Obtaining Digital Signature and Public-Key Cryptosystems." Communication of the ACM, 21, 2, Feb 1978, 120-126.

10. RSA Laboratories, "The RSA Laboratories Secret-key Challenge: Cryptographic Challenges" , www.rsasecurity.com/rsalabs/node.asp

11. Daemon, J. and Rijmen, V. "The Rijndael Block Cipher: AES Proposal", NIST, Version 2, March 1999.

12. Daemon, J., and Rijmen, V. "Rijndael: The Advanced Encryption Standard." Dr. Dobb's Journal, **26**, 3, March 2001, 137-139.

13. Daemon, J., and Rijmen, V. "The Design of Rijndael: The Wide Trail Strategy Explained." New York, Springer – Verlag, 2000.

14. National Policy on the Use of the Advanced Encryption Standard (AES) to Protect National Security Systems and National Security Information, CNSS Policy No. 1, Fact Sheet No. 1, June 2003, www.nstissc.gov/Assets/pdf/fact%20sheet.pdf

15. Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. In *Proc. 2nd AES candidate conference*, pp 15–34, NIST, 1999, www.macfergus.com/pub/icrijndael.html

16. Lidl, R., and Niederreiter, H. "Introduction to finite fields and their applications" Cambridge University Press, 1986.

17. Brunner, H., Curiger, A., Hofstetter, M. "On Computing Multiplicative Inverse in GF(2^m)." IEEE Transactions On Computers, **42**,8, August 1993, 1010-1015

18. Morioka, S., Katayama, Y. "Iterative Algorithm for Multiplicative Inversion in GF(2^m)." ISIT 2000, Sorrento, Italy, 25-30 June, 2000.

19. Bell, C., "Ultracomputer: A teraflop before its time", Communication of the ACM, 35, 8, 27-47, 1992

20. Flynn, M., "Some Computer Organizations and their effectiveness.", IEEE Transactions on Computer, C-21, 9, 948-960, 1972

APPENDICES

PROGRAM LISTING

A Program Listing for Serial Implementation

Table.h

```
#include <string.h>
int Nb = 4; // words in a block, always 4 for now


unsigned char E[256]; // "exp" table (base 0x03)
unsigned char L[256]; // "Log" table (base 0x03)
unsigned char S[256]; // SubBytes table
unsigned char invS[256]; // inverse of SubBytes table
unsigned char inv[256]; // multiplicative inverse table
unsigned char powX[256];// powers of x = 0x02
char dig[] = {'0','1','2','3','4','5','6','7',
              '8','9','a','b','c','d','e','f'};
char temp[2];

int subBytes(unsigned char b);
unsigned char FFInv(unsigned char b);
unsigned char SBox(unsigned char b);
unsigned char invSBox(unsigned char b);
unsigned char Rcon(int i);
unsigned char FFMulFast(unsigned char a, unsigned char b);
unsigned char FFMul(unsigned char a, unsigned char b);
void loadE();
void loadL();
void loadS();
void loadInv();
void loadInvS();
void loadPowX();
int ithBit(unsigned char b, int i);
int subBytes(unsigned char b);
void hex(unsigned char a)
void printArray(char name[], int round, unsigned char a[]);
void printArray(char name[], int round, unsigned char s[][4]);

// Routines to access table entries
 unsigned char SBox(unsigned char b) {
      return S[b & 0xff];
}

 unsigned char invSBox(unsigned char b) {
      return invS[b & 0xff];
}
 unsigned char Rcon(int i) {
      return powX[i-1];
}
```

```
// FFMulFast: fast multiply using table lookup
unsigned char FFMulFast(unsigned char a, unsigned char b){
        int t = 0;;

        if (a == 0 || b == 0)
                return 0;
        t = (L[(a & 0xff)] & 0xff) + (L[(b & 0xff)] & 0xff);
         if (t > 255)
                 t = t - 255;
        return E[(t & 0xff)];
}

// FFMul: slow multiply, using shifting
 unsigned char FFMul(unsigned char a, unsigned char b) {

        unsigned char aa = a, bb = b, r = 0, t;
        while (aa != 0) {
                if ((aa & 1) != 0)
                        r = (unsigned char)(r ^ bb);
                t = (unsigned char)(bb & 0x80);
                bb = (unsigned char)(bb << 1);

                if (t != 0)
                        bb = (unsigned char)(bb ^ 0x1b);
                aa = (unsigned char)((aa & 0xff) >> 1);
        }
        return r;
}

// loadE: create and load the E table
void loadE() {

        unsigned char x = (unsigned char)0x01;
        int index = 0;
        E[index++] = (unsigned char)0x01;

        for (int i = 0; i < 255; i++) {
                unsigned char y = FFMul(x, (unsigned char)0x03);
                E[index++] = y;
                x = y;
        }
}

// loadL: load the L table using the E table
void loadL() { // careful: had 254 below several places

         for (int i = 0; i < 255; i++) {
                L[E[i] & 0xff] = (unsigned char)i;
         }
}

/ loadS: load in the table S
void loadS() {

        for (int i = 0; i < 256; i++)
                S[i] = (unsigned char)(subBytes((unsigned char)(i & 0xff)) & 0xff);
}
```

```
// loadInv: load in the table inv
  void loadInv() {

         for (int i = 0; i < 256; i++)
               inv[i] = (unsigned char)(FFInv((unsigned char)(i & 0xff)) & 0xff);
  }

  // loadInvS: load the invS table using the S table
  void loadInvS() {

         for (int i = 0; i < 256; i++) {
               invS[S[i] & 0xff] = (unsigned char)i;
    }
  }

  // loadPowX: load the powX table using multiplication
 void loadPowX() {

         unsigned char x = (unsigned char)0x02;
         unsigned char xp = x;
         powX[0] = 1; powX[1] = x;

         for (int i = 2; i < 15; i++) {
               xp = FFMul(xp, x);
               powX[i] = xp;
         }
  }

  // FFInv: the multiplicative inverse of a byte value
   unsigned char FFInv(unsigned char b) {

         unsigned char e = L[b & 0xff];
         return E[0xff - (e & 0xff)];
  }

  // ithBIt: return the ith bit of a byte
   int ithBit(unsigned char b, int i) {

         int m[] = {0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80};
         return  (b & m[i]) >> i;
  }

  // subBytes: the subBytes function
   int subBytes(unsigned char b) {

         unsigned char inB = b;
         int res = 0;

         if (b != 0) // if b == 0, leave it alone
               b = (unsigned char)(FFInv(b) & 0xff);

         unsigned char c = (unsigned char)0x63;
         for (int i = 0; i < 8; i++) {
               int temp = 0;
               temp = ithBit(b, i) ^ ithBit(b, (i+4)%8) ^ ithBit(b, (i+5)%8) ^
                        ithBit(b, (i+6)%8) ^ ithBit(b, (i+7)%8) ^ ithBit(c, i);
```

```
            res = res | (temp << i);
        }
        return res;
}

// hex: print a byte as two hex digits
void hex(unsigned char a) {
      temp[0] = dig[(a & 0xff) >> 4];
      temp[1] = dig[a & 0x0f];
}

void printArray(char name[], int round, unsigned char a[]) {
      printf("%s %d: ",name, round);
      for (int i = 0; i < 16; i++){
            hex(a[i]);
            printf("%c%c ", temp[0], temp[1]);
      }
      printf("\n");
}
void printArray(char name[], int round, unsigned char s[][4]) {
       static int count=0;
       if (round >=10){
            count++;
            if (!strcmp(name,"PlainText"))
                  printf("%s%d\t: ",name, round);
            else
                  printf("%s%d\t\t: ",name, round);
       }
       else
             printf("%s%d\t\t: ",name, round);
       for (int c = 0; c < Nb; c++)
            for (int r = 0; r < 4; r++){
                  hex(s[r][c]);
                  if (temp[0]>=97 && temp[0]<=102)
                        temp[0] -= 32;
                  if (temp[1]>=97 && temp[1]<=102)
                        temp[1] -= 32;
                  printf("%c%c ", temp[0], temp[1]);
            }
            printf("\n");
}
```

AESCipher.cpp

```
#include <time.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/timeb.h>
#include <string.h>
#include "Table.h"



int Nk, NkIn; // key length in words
int Nr; // number of rounds, = Nk + 6
int wCount; // position in w for RoundKey (= 0 each encrypt)
unsigned char *w; // the expanded key
unsigned     char key[16];
```

```
void KeyExpansion(unsigned char key[], unsigned char w[]);
void MixColumns(unsigned char s[][4]);
void ShiftRows(unsigned char state[][4]);
void SubBytesTransformation(unsigned char state[][4]);
void AddRoundKey(unsigned char state[][4]);
void FileRead(unsigned char state[][4], unsigned char key[]);
```

//FileRead: Reading the plain text from the input file

```
void FileRead(unsigned char state[][4], unsigned char key[]){

        int count =0;
        unsigned int temp;
        FILE *fptrR;
        unsigned char block[16];

        if ((fptrR = fopen("input.txt","r")) == NULL)
                printf("File cannot be opened");
        else{

                for(;!feof(fptrR) && count < 16 ; ++count){
                        fscanf(fptrR,"%X", &temp);
                        block[count] = temp;
                }
                count = 0;
                for(;!feof(fptrR) && count < 16 ; ++count){
                        fscanf(fptrR,"%X", &temp);
                        key[count] = temp;
                }
                count =0;
                for(int column=0; column<4;column++)
                        for(int row =0; row<4; row++)
                                state[row][column] = block[count++];
        }
    }
```

// KeyExpansion: expand key, byte-oriented code, but tracks words

```
 void KeyExpansion(unsigned char key[], unsigned char w[]) {
       unsigned char temp[4];

       // first just copy key to w
       int j = 0,count=0;
       while (j < 4*Nk)
              w[j] = key[j++];

       // here j == 4*Nk;
       int i;

       while(j < 4*Nb*(Nr+1)) {
              i = j/4; // j is always multiple of 4 here

              // handle everything word-at-a time, 4 bytes at a time
              for (int iTemp = 0; iTemp < 4; iTemp++)
                     temp[iTemp] = w[j-4+iTemp];
              if (i % Nk == 0) {
```

```
                        unsigned char ttemp, tRcon;
                        unsigned char oldtemp0 = temp[0];

                        for (int iTemp = 0; iTemp < 4; iTemp++) {
                                if (iTemp == 3)
                                        ttemp = oldtemp0;
                                else
                                        ttemp = temp[iTemp+1];

                                if (iTemp == 0)
                                        tRcon = Rcon(i/Nk);
                                else
                                        tRcon = 0;
                                temp[iTemp] = (unsigned char)(SBox(ttemp) ^ tRcon);
                        }
                }

                else if (Nk > 6 && (i%Nk) == 4) {
                        for (iTemp = 0; iTemp < 4; iTemp++)
                                temp[iTemp] = SBox(temp[iTemp]);
                }

                for (iTemp = 0; iTemp < 4; iTemp++)
                        w[j+iTemp] = (unsigned char)(w[j - 4*Nk + iTemp] ^
temp[iTemp]);
                 j = j + 4;
        }
  }

 // SubBytes: apply Sbox substitution to each byte of state
 void SubBytesTransformation(unsigned char state[][4]) {

        for (int row = 0; row < 4; row++)
              for (int col = 0; col < Nb; col++)
                        state[row][col] = SBox(state[row][col]);
 }

 // ShiftRows: left circular shift of rows 1, 2, 3 by 1, 2, 3
 void ShiftRows(unsigned char state[][4]) {

         unsigned char t[4];

         for (int r = 1; r < 4; r++) {
               for (int c = 0; c < Nb; c++)
                        t[c] = state[r][(c + r)%Nb];
               for (c = 0; c < Nb; c++)
                        state[r][c] = t[c];
         }
 }

 // MixColumns: complex and sophisticated mixing of columns
 void MixColumns(unsigned char s[][4]) {

        int sp[4];
        unsigned char b02 = (unsigned char)0x02, b03 = (unsigned char)0x03;

        for (int c = 0; c < 4; c++) {
```

```
            sp[0] = FFMul(b02, s[0][c]) ^ FFMul(b03, s[1][c]) ^
                        s[2][c] ^ s[3][c];

            sp[1] = s[0][c] ^ FFMul(b02, s[1][c]) ^
                              FFMul(b03, s[2][c]) ^  s[3][c];

            sp[2] = s[0][c] ^  s[1][c]  ^
                              FFMul(b02, s[2][c]) ^ FFMul(b03, s[3][c]);

            sp[3] = FFMul(b03, s[0][c]) ^ s[1][c]  ^
                                s[2][c] ^ FFMul(b02, s[3][c]);

            for (int i = 0; i < 4; i++)
                    s[i][c] = (unsigned char)(sp[i]);
      }
}
```

// AddRoundKey: xor a portion of expanded key with state

```
void AddRoundKey(unsigned char state[][4]) {

       for (int c = 0; c < Nb; c++)
    for (int r = 0; r < 4; r++)
      state[r][c] = (unsigned char)(state[r][c] ^ w[wCount++]);
}
```

// SubBytes: apply inverse Sbox substitution to each byte of state

```
void InvSubBytes(unsigned char state[][4]) {

       for (int row = 0; row < 4; row++)
            for (int col = 0; col < Nb; col++)
                  state[row][col] = invSBox(state[row][col]);
}
```

// ShiftRows: right circular shift of rows 1, 2, 3 by 1, 2, 3

```
void InvShiftRows(unsigned char state[][4]) {

      unsigned char t[4];
      for (int r = 1; r < 4; r++) {
            for (int c = 0; c < Nb; c++)
                  t[(c + r)%Nb] = state[r][c];

            for (c = 0; c < Nb; c++)
                  state[r][c] = t[c];
      }
}
```

// InvMixColumns: complex and sophisticated mixing of columns

```
void InvMixColumns(unsigned char s[][4]) {

      int sp[4];
      unsigned char b0b = (unsigned char)0x0b;
      unsigned char b0d = (unsigned char)0x0d;
      unsigned char b09 = (unsigned char)0x09;
      unsigned char b0e = (unsigned char)0x0e;

      for (int c = 0; c < 4; c++) {
            sp[0] = FFMul(b0e, s[0][c]) ^ FFMul(b0b, s[1][c]) ^
```

```
                        FFMul(b0d, s[2][c]) ^ FFMul(b09, s[3][c]);

                sp[1] = FFMul(b09, s[0][c]) ^ FFMul(b0e, s[1][c]) ^
                        FFMul(b0b, s[2][c]) ^ FFMul(b0d, s[3][c]);

                sp[2] = FFMul(b0d, s[0][c]) ^ FFMul(b09, s[1][c]) ^
                        FFMul(b0e, s[2][c]) ^ FFMul(b0b, s[3][c]);

                sp[3] = FFMul(b0b, s[0][c]) ^ FFMul(b0d, s[1][c]) ^
                        FFMul(b09, s[2][c]) ^ FFMul(b0e, s[3][c]);
                for (int i = 0; i < 4; i++)
                        s[i][c] = (unsigned char)(sp[i]);
        }
}
```

// InvAddRoundKey: same as AddRoundKey, but backwards

```
void InvAddRoundKey(unsigned char state[][4]) {

        for (int c = Nb - 1; c >= 0; c--)
                for (int r = 3; r >= 0 ; r--)
                        state[r][c] = (unsigned char)(state[r][c] ^ w[--wCount]);
        }
```

void main(){

```
        char tmpbuf[128],tmpbuf1[128];
        struct _timeb tstruct,tstruct1;
        int round = 1;

        printf("Enter words in Key: ");
        scanf("%d",&NkIn);
        Nk = NkIn; // words in a key, = 4, or 6, or 8
        Nr = Nk + 6; // corresponding number of rounds
        w = new unsigned char[4*Nb*(Nr+1)]; // room for expanded key

        wCount = 0; // count bytes in expanded key throughout encryption
        unsigned char state[4][4]; // the state array

        FileRead(state, key);
        loadE();
        loadL();
        loadS();
        loadInv();
        loadInvS();
        loadPowX();
        KeyExpansion(key, w); // length of w depends on Nr
```

//Start time of encryption

```
        _strtime( tmpbuf );
        printf( "Encryption Start At: ");
        for(int i = 0; i <8;i++)
                        printf( "%c", tmpbuf[i] );
        _ftime( &tstruct );
        printf( ":%u\n\n", tstruct.millitm );
```

//Encryption

```
 printf("Encrypting...\n");
 printArray("Round",round, state);
 AddRoundKey(state); // xor with expanded key
 for (; round < Nr; round++) {
        SubBytesTransformation(state); // S-box substitution
        printArray("Round",round, state);
        ShiftRows(state); // mix up rows
        printArray("Round",round, state);
        MixColumns(state); // complicated mix of columns
        printArray("Round",round, state);
        AddRoundKey(state); // xor with expanded key
        printArray("Round",round, state);
 }

printArray("Round" , Nr , state);
SubBytesTransformation(state); // S-box substitution
printArray("Round" , Nr , state);
ShiftRows(state); // mix up rows
printArray("Round" , Nr , state);
AddRoundKey(state);
 printArray("CipherText" , Nr , state);

 //End time of encryption
 _strtime( tmpbuf1 );
 printf( "\nEncryption End At: ");
 for(i = 0; i <8;i++)
                printf( "%c", tmpbuf1[i] );
 _ftime( &tstruct1 );
 printf( ":%u\n\n", tstruct1.millitm );

//Start time of decryption
 _strtime( tmpbuf );
 printf( "Decryption Start At: ");
 for(i = 0; i <8;i++)
                printf( "%c", tmpbuf[i] );
 _ftime( &tstruct );
 printf( ":%u\n\n", tstruct.millitm );

 //Decryption
 printf("Decrypting...\n");
 wCount = 4*Nb*(Nr+1); // count bytes during decryption

 printArray("Round",1, state);
 InvAddRoundKey(state); // xor with expanded key

 for (round = Nr-1; round >= 1; round--) {
        printArray("Round",(Nr - round), state);
        InvShiftRows(state); // mix up rows
        printArray("Round",(Nr - round), state);
        InvSubBytes(state); // inverse S-box substitution
        printArray("Round",(Nr - round), state);
        InvAddRoundKey(state); // xor with expanded key
        printArray("Round",(Nr - round), state);
        InvMixColumns(state); // complicated mix of columns
 }

printArray("Round", Nr , state);
```

```
        InvShiftRows(state); // mix up rows
        printArray("Round",Nr, state);
        InvSubBytes(state); // inverse S-box substitution
        InvAddRoundKey(state); // xor with expanded key
        printArray("Round",Nr, state);
        printArray("PlainText" , Nr , state);

        //End time of deccryption
        _strtime( tmpbuf1 );
        printf( "\nDeccryption End At: ");
        for(i = 0; i <8;i++)
                        printf( "%c", tmpbuf1[i] );
        _ftime( &tstruct1 );
        printf( ":%u\n\n", tstruct1.millitm );

}
```

B Program Listing for Parallel Implementation

```
#include <stdio.h>
#include <mpi.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <string.h>


#define ArraySize     160
#define NkIn  4

  int Nb = 4; // words in a block, always 4 for now
  unsigned char E[256]; // "exp" table (base 0x03)
  unsigned char L[256]; // "Log" table (base 0x03)
  unsigned char S[256]; // SubBytes table
  unsigned char invS[256]; // inverse of SubBytes table
  unsigned char inv[256]; // multiplicative inverse table
  unsigned char powX[256];// powers of x = 0x02
  char dig[] = {'0','1','2','3','4','5','6','7',
                                                '8','9','a','b','c','d','e','f'};
  char temp[2];

  int Nk, NkIn; // key length in words
  int Nr; // number of rounds, = Nk + 6
  int wCount; // position in w for RoundKey (= 0 each encrypt)
  unsigned char *w; // the expanded key
  unsigned     char key[16];
  unsigned char *my_pointer[50], k =0,i = 1;
  int my_rank;
  MPI_Status  status;

  int subBytes(unsigned char b);
  unsigned char FFInv(unsigned char b);
  unsigned char SBox(unsigned char b);
  unsigned char invSBox(unsigned char b);
  unsigned char Rcon(int i);
  unsigned char FFMulFast(unsigned char a, unsigned char b);
  unsigned char FFMul(unsigned char a, unsigned char b);
  void loadE();
  void loadL();
  void loadS();
  void loadInv();
  void loadPowX();
  unsigned char FFInv(unsigned char b);
  int ithBit(unsigned char b, int i);
  int subBytes(unsigned char b);
  void hex(unsigned char a);
  void printArray(char name[], int round, unsigned char a[]);
  void printArray(char name[], int round, unsigned char s[][4]);
  void KeyExpansion(unsigned char key[], unsigned char w[]);
  void MixColumns(unsigned char s[][4]);
  void ShiftRows(unsigned char state[][4]);
  void SubBytesTransformation(unsigned char state[][4]);
```

```
void AddRoundKey(unsigned char state[][4]);
void InvAddRoundKey(unsigned char state[][4]);
void InvMixColumns(unsigned char s[][4]);
void InvShiftRows(unsigned char state[][4]);
void InvSubBytes(unsigned char state[][4]);
void Encryption(unsigned char blocks[] );
void KeyRead(unsigned char key[]);
void Decryption(unsigned char blocks[]);
void CreateStateMatrix(unsigned char block[],unsigned char state[][4]);
void MergeEncryptedData(unsigned char Array[], unsigned ArrayTwo[],
                        unsigned char ArrayMain[], int FHIndex, int SHIndex);
void BTM(int low, int high);


void main(){

        int p;
        unsigned char Array[ArraySize];
        int n;
        int count,ff, rank;
        FILE *fp;
        double startwtime = 0.0, endwtime;

        MPI_Init (&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
        MPI_Comm_size(MPI_COMM_WORLD,&p);

        Nk = NkIn; // words in a key, = 4, or 6, or 8
        Nr = Nk + 6; // corresponding number of rounds
        w = new unsigned char[4*Nb*(Nr+1)]; // room for expanded key
        srand(time(0));

        for(int count = 0; count<16; count++)
               key[count] = rand() % 256;

        loadE();
        loadL();
        loadS();
        loadInv();
        loadInvS();
        loadPowX();
        KeyExpansion(key, w); // length of w depends on Nr

        my_pointer[k] = new unsigned char[ArraySize];
        for(int count = 0; count < ArraySize; count++)
                        my_pointer[k][count] = rand() % 256;



      if (my_rank == 0){
                fprintf(fp,"Process 0: ");
                for(ff = 0; ff < ArraySize;ff++){
                        if (ff % 16 == 0)
                                fprinf(fp,"\n");
                        fprintf(fp,"%X ", my_pointer[0][ff]);
                }
```

```
        for(rank = 1; rank <p; rank++){
                MPI_Recv(Array, ArraySize, MPI_UNSIGNED_CHAR,rank,
                                        20, MPI_COMM_WORLD, &status);
                fprintf(fp,"\nProcess %d", rank);
                for(ff = 0;; ff<ArraySize; ff++){
                        if (ff % 16 == 0)
                                fprintf(fp,"\n");
                        fprintf(fp,"%X ", Array[ff]);
                }
        }
}
else
        MPI_Send(mp_pointer[0], ArraySize, MPI_UNSIGNED_CHAR,0,20,
                                                MPI_COMM_WORLD);

//Encryption
printf("Encrypting...\n");
startwtime = MPI_Wtime();
Encryption(my_pointer[k]);
BTM(0,p-1);
endwtime = MPI_Wtime();

if (my_rank == 0){
        fprintf("fp","\nEncrypted Data: ");
        for(count = 0;count<ArraySize * p; count++){
                if (count % 16 == 0)
                        fprintf(fp,"\n");
                printf("%X ", my_pointer[k][count]);
                fprintf(fp,"%X ", my_pointer[k][count]);
        }
        printf("\nTotal time to encrypt: %f",
                                                (endwtime - startwtime));
}


//Decryption
printf("Decrypting...\n");
startwtime = MPI_Wtime();
// In each processor, index 1 of my_pointer contains
// the encryptrd data
Decryption(my_pointer[1]);
BTM(0,p-1);
endwtime = MPI_Wtime();



if (my_rank == 0){
        fprintf("fp","\nDecrypted Data: ");
        for(count = 0;count<ArraySize * p; count++){
                if (count % 16 == 0)
                        fprintf(fp,"\n");
                printf("%X ", my_pointer[k][count]);
                fprintf(fp,"%X ", my_pointer[k][count]);
        }
        printf("\nTotal time to decrypt: %f",
                                (endwtime - startwtime));
}
```

```
        MPI_Finalize();

    }

// Routines to access table entries
    unsigned char SBox(unsigned char b) {
        return S[b & 0xff];
    }

    unsigned char invSBox(unsigned char b) {
        return invS[b & 0xff];
    }
    unsigned char Rcon(int i) {
        return powX[i-1];
    }

    // FFMulFast: fast multiply using table lookup
    unsigned char FFMulFast(unsigned char a, unsigned char b){
        int t = 0;;

        if (a == 0 || b == 0)
            return 0;
        t = (L[(a & 0xff)] & 0xff) + (L[(b & 0xff)] & 0xff);
        if (t > 255)
            t = t - 255;
        return E[(t & 0xff)];
    }

    // FFMul: slow multiply, using shifting
    unsigned char FFMul(unsigned char a, unsigned char b) {

        unsigned char aa = a, bb = b, r = 0, t;
        while (aa != 0) {
            if ((aa & 1) != 0)
                r = (unsigned char)(r ^ bb);
            t = (unsigned char)(bb & 0x80);
            bb = (unsigned char)(bb << 1);

            if (t != 0)
                bb = (unsigned char)(bb ^ 0x1b);
            aa = (unsigned char)((aa & 0xff) >> 1);
        }
        return r;
    }

    // loadE: create and load the E table
    void loadE() {

        unsigned char x = (unsigned char)0x01;
        int index = 0;
        E[index++] = (unsigned char)0x01;

        for (int i = 0; i < 255; i++) {
            unsigned char y = FFMul(x, (unsigned char)0x03);
            E[index++] = y;
            x = y;
        }
```

```
  }

// loadL: load the L table using the E table
  void loadL() { // careful: had 254 below several places

          for (int i = 0; i < 255; i++) {
                L[E[i] & 0xff] = (unsigned char)i;
          }
  }

/ loadS: load in the table S
void loadS() {

          for (int i = 0; i < 256; i++)
                S[i] = (unsigned char)(subBytes((unsigned char)(i & 0xff)) & 0xff);
  }

// loadInv: load in the table inv
  void loadInv() {

          for (int i = 0; i < 256; i++)
                inv[i] = (unsigned char)(FFInv((unsigned char)(i & 0xff)) & 0xff);
  }

  // loadInvS: load the invS table using the S table
  void loadInvS() {

          for (int i = 0; i < 256; i++) {
                invS[S[i] & 0xff] = (unsigned char)i;
    }
  }

  // loadPowX: load the powX table using multiplication
 void loadPowX() {

          unsigned char x = (unsigned char)0x02;
          unsigned char xp = x;
          powX[0] = 1; powX[1] = x;

          for (int i = 2; i < 15; i++) {
                xp = FFMul(xp, x);
                powX[i] = xp;
          }
  }

  // FFInv: the multiplicative inverse of a byte value
   unsigned char FFInv(unsigned char b) {

          unsigned char e = L[b & 0xff];
          return E[0xff - (e & 0xff)];
  }

  // ithBIt: return the ith bit of a byte
   int ithBit(unsigned char b, int i) {

          int m[] = {0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80};
          return  (b & m[i]) >> i;
```

```
}

// subBytes: the subBytes function
 int subBytes(unsigned char b) {

        unsigned char inB = b;
        int res = 0;

        if (b != 0) // if b == 0, leave it alone
                b = (unsigned char)(FFInv(b) & 0xff);

        unsigned char c = (unsigned char)0x63;
        for (int i = 0; i < 8; i++) {
                int temp = 0;
                temp = ithBit(b, i) ^ ithBit(b, (i+4)%8) ^ ithBit(b, (i+5)%8) ^
                        ithBit(b, (i+6)%8) ^ ithBit(b, (i+7)%8) ^ ithBit(c, i);
                res = res | (temp << i);
        }
        return res;
}

// hex: print a byte as two hex digits
void hex(unsigned char a) {
      temp[0] = dig[(a & 0xff) >> 4];
      temp[1] = dig[a & 0x0f];
}

void printArray(char name[], int round, unsigned char a[]) {
      printf("%s %d: ",name, round);
      for (int i = 0; i < 16; i++){
                hex(a[i]);
                printf("%c%c ", temp[0], temp[1]);
      }
      printf("\n");
}
void printArray(char name[], int round, unsigned char s[][4]) {
        static int count=0;
        if (round >=10){
                count++;
                if (!strcmp(name,"PlainText"))
                        printf("%s%d\t: ",name, round);
                else
                        printf("%s%d\t\t: ",name, round);
        }
        else
               printf("%s%d\t\t: ",name, round);
        for (int c = 0; c < Nb; c++)
                for (int r = 0; r < 4; r++){
                        hex(s[r][c]);
                        if (temp[0]>=97 && temp[0]<=102)
                                temp[0] -= 32;
                        if (temp[1]>=97 && temp[1]<=102)
                                temp[1] -= 32;
                        printf("%c%c ", temp[0], temp[1]);
                }
                printf("\n");
}
```

AESCipher.cpp

```
#include <time.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/timeb.h>
#include <string.h>
#include "Table.h"



  int Nk, NkIn; // key length in words
  int Nr; // number of rounds, = Nk + 6
  int wCount; // position in w for RoundKey (= 0 each encrypt)
  unsigned char *w; // the expanded key
  unsigned     char key[16];


  void KeyExpansion(unsigned char key[], unsigned char w[]);
  void MixColumns(unsigned char s[][4]);
  void ShiftRows(unsigned char state[][4]);
  void SubBytesTransformation(unsigned char state[][4]);
  void AddRoundKey(unsigned char state[][4]);
  void FileRead(unsigned char state[][4], unsigned char key[]);

  //FileRead: Reading the plain text from the input file
  void FileRead(unsigned char state[][4], unsigned char key[]){

              int count =0;
              unsigned int temp;
              FILE *fptrR;
              unsigned char block[16];

              if ((fptrR = fopen("input.txt","r")) == NULL)
                      printf("File cannot be opened");
              else{

                      for(;!feof(fptrR) && count < 16 ; ++count){
                              fscanf(fptrR,"%X", &temp);
                              block[count] = temp;
                      }
                      count = 0;
                      for(;!feof(fptrR) && count < 16 ; ++count){
                              fscanf(fptrR,"%X", &temp);
                              key[count] = temp;
                      }
                      count =0;
                      for(int column=0; column<4;column++)
                              for(int row =0; row<4; row++)
                                      state[row][column] = block[count++];
              }
      }


  // KeyExpansion: expand key, byte-oriented code, but tracks words
   void KeyExpansion(unsigned char key[], unsigned char w[]) {
         unsigned char temp[4];
```

```
        // first just copy key to w
        int j = 0,count=0;
        while (j < 4*Nk)
                w[j] = key[j++];

        // here j == 4*Nk;
        int i;

        while(j < 4*Nb*(Nr+1)) {
                i = j/4; // j is always multiple of 4 here

                // handle everything word-at-a time, 4 bytes at a time
                for (int iTemp = 0; iTemp < 4; iTemp++)
                        temp[iTemp] = w[j-4+iTemp];
                if (i % Nk == 0) {
                        unsigned char ttemp, tRcon;
                        unsigned char oldtemp0 = temp[0];

                        for (int iTemp = 0; iTemp < 4; iTemp++) {
                                if (iTemp == 3)
                                        ttemp = oldtemp0;
                                else
                                        ttemp = temp[iTemp+1];

                                if (iTemp == 0)
                                        tRcon = Rcon(i/Nk);
                                else
                                        tRcon = 0;
                                temp[iTemp] = (unsigned char)(SBox(ttemp) ^ tRcon);
                        }
                }

                else if (Nk > 6 && (i%Nk) == 4) {
                        for (iTemp = 0; iTemp < 4; iTemp++)
                                temp[iTemp] = SBox(temp[iTemp]);
                }

                for (iTemp = 0; iTemp < 4; iTemp++)
                        w[j+iTemp] = (unsigned char)(w[j - 4*Nk + iTemp] ^
temp[iTemp]);
                j = j + 4;
        }
  }

  // SubBytes: apply Sbox substitution to each byte of state
  void SubBytesTransformation(unsigned char state[][4]) {

        for (int row = 0; row < 4; row++)
                for (int col = 0; col < Nb; col++)
                        state[row][col] = SBox(state[row][col]);
  }

  // ShiftRows: left circular shift of rows 1, 2, 3 by 1, 2, 3
  void ShiftRows(unsigned char state[][4]) {

         unsigned char t[4];
```

```
        for (int r = 1; r < 4; r++) {
                for (int c = 0; c < Nb; c++)
                        t[c] = state[r][(c + r)%Nb];
                for (c = 0; c < Nb; c++)
                        state[r][c] = t[c];
        }
}

 // MixColumns: complex and sophisticated mixing of columns
void MixColumns(unsigned char s[][4]) {

        int sp[4];
        unsigned char b02 = (unsigned char)0x02, b03 = (unsigned char)0x03;

        for (int c = 0; c < 4; c++) {
               sp[0] = FFMul(b02, s[0][c]) ^ FFMul(b03, s[1][c]) ^
                               s[2][c] ^ s[3][c];

                sp[1] = s[0][c] ^ FFMul(b02, s[1][c]) ^
                                FFMul(b03, s[2][c]) ^  s[3][c];

                sp[2] = s[0][c] ^  s[1][c]  ^
                                FFMul(b02, s[2][c]) ^ FFMul(b03, s[3][c]);

                sp[3] = FFMul(b03, s[0][c]) ^ s[1][c]  ^
                                  s[2][c] ^ FFMul(b02, s[3][c]);

                for (int i = 0; i < 4; i++)
                        s[i][c] = (unsigned char)(sp[i]);
        }
}

 // AddRoundKey: xor a portion of expanded key with state
 void AddRoundKey(unsigned char state[][4]) {

        for (int c = 0; c < Nb; c++)
     for (int r = 0; r < 4; r++)
       state[r][c] = (unsigned char)(state[r][c] ^ w[wCount++]);
}

 // SubBytes: apply inverse Sbox substitution to each byte of state
 void InvSubBytes(unsigned char state[][4]) {

        for (int row = 0; row < 4; row++)
               for (int col = 0; col < Nb; col++)
                        state[row][col] = invSBox(state[row][col]);
}

 // ShiftRows: right circular shift of rows 1, 2, 3 by 1, 2, 3
 void InvShiftRows(unsigned char state[][4]) {

        unsigned char t[4];
        for (int r = 1; r < 4; r++) {
                for (int c = 0; c < Nb; c++)
                        t[(c + r)%Nb] = state[r][c];
```

```
            for (c = 0; c < Nb; c++)
                    state[r][c] = t[c];
        }
}

// InvMixColumns: complex and sophisticated mixing of columns
void InvMixColumns(unsigned char s[][4]) {

        int sp[4];
        unsigned char b0b = (unsigned char)0x0b;
        unsigned char b0d = (unsigned char)0x0d;
        unsigned char b09 = (unsigned char)0x09;
        unsigned char b0e = (unsigned char)0x0e;

        for (int c = 0; c < 4; c++) {
              sp[0] = FFMul(b0e, s[0][c]) ^ FFMul(b0b, s[1][c]) ^
                            FFMul(b0d, s[2][c]) ^ FFMul(b09, s[3][c]);

              sp[1] = FFMul(b09, s[0][c]) ^ FFMul(b0e, s[1][c]) ^
                            FFMul(b0b, s[2][c]) ^ FFMul(b0d, s[3][c]);

              sp[2] = FFMul(b0d, s[0][c]) ^ FFMul(b09, s[1][c]) ^
                            FFMul(b0e, s[2][c]) ^ FFMul(b0b, s[3][c]);

              sp[3] = FFMul(b0b, s[0][c]) ^ FFMul(b0d, s[1][c]) ^
                            FFMul(b09, s[2][c]) ^ FFMul(b0e, s[3][c]);
              for (int i = 0; i < 4; i++)
                    s[i][c] = (unsigned char)(sp[i]);
      }
}

// InvAddRoundKey: same as AddRoundKey, but backwards
void InvAddRoundKey(unsigned char state[][4]) {

         for (int c = Nb - 1; c >= 0; c--)
              for (int r = 3; r >= 0 ; r--)
                      state[r][c] = (unsigned char)(state[r][c] ^ w[--wCount]);
      }

 void CreateStateMatrix(unsigned char block[],unsigned char state[][4]){
         int count =0;
         for(int column=0; column<4; column++)
              for(int row=0; row<4;row++)
                    state[row][column] = block[count++] ;
 }

void Encryption(unsigned char blocks[] ){
        int round = 1;
        unsigned char state[4][4]; // the state array
        int column;
        unsigned char block[16];

        //Creating the spaces for storing the encrypted data into
        //the index no 1 of my_pointer. Current value of k = 0;
        my_pointer[++k] = new unsigned char[ArraySize];
        int toatlNoElement = 0;
```

```
	for(int row = 0; row<ArraySize;row +=16){
		for(column = 0; column<16; column++){
			block[column] = blocks[row+column];
		}

		CreateStateMatrix(block,state);
		wCount = 0;
		printArray("Round",round, state);
		AddRoundKey(state); // xor with expanded key
		for (; round < Nr; round++) {
			SubBytesTransformation(state); // S-box substitution
			ShiftRows(state); // mix up rows
			MixColumns(state); // complicated mix of columns
			AddRoundKey(state); // xor with expanded key
		}

		SubBytesTransformation(state); // S-box substitution
		ShiftRows(state); // mix up rows
		AddRoundKey(state);
		printArray("CipherText" , Nr , state);

		//store the encrypted data into the index no 1 of my_pointer
		for(int column= 0;column<Nb;column++)
		    for(int row = 0; row<Nb;row++);
			my_pointer[k][toatlNoElement++] = state[row][column];
	}
}
```

void Decryption(unsigned char blocks[]){

```
	int round=1;
	unsigned char state[4][4]; // the state array
	int column;
	unsigned char block[16];

	//Creating the spaces for storing the plainText data into
	//the index no k+1 of my_pointer.
	my_pointer[++k] = new unsigned char[ArraySize];
	int toatlNoElement = 0;

	for(int row = 0; row<ArraySize;row +=16){
		for(column = 0; column<16; column++){
			block[column] = blocks[row+column];
		}

		CreateStateMatrix(block,state);
		wCount = 4*Nb*(Nr+1); // count bytes during decryption
		printArray("Round",1, state);
		InvAddRoundKey(state); // xor with expanded key

		for (round = Nr-1; round >= 1; round--) {
			InvShiftRows(state); // mix up rows
			InvSubBytes(state); // inverse S-box substitution
			InvAddRoundKey(state); // xor with expanded key
			InvMixColumns(state); // complicated mix of columns
		}
		InvShiftRows(state); // mix up rows
```

```
            InvSubBytes(state); // inverse S-box substitution
            InvAddRoundKey(state); // xor with expanded key
            printArray("PlainText" , Nr , state);

            //store the plaintext data into the index no k+1 of my_pointer
            for(int column= 0;column<Nb;column++)
                for(int row = 0; row<Nb;row++);
                    my_pointer[k][toatlNoElement++] = state[row][column];
        }
}


void MergeEncryptedData(unsigned char Array[], unsigned char
            ArrayTwo[], unsigned char ArrayMain[],
                                        int FHIndex, int SHIndex){

        int numberOfData = 0;
        for(int count =0; count<= FHIndex;count++)
                ArrayMain[numberOfData++] = Array[count];

        for(count = 0; count<= SHIndex; count++)
                ArrayMain[numberOfData++] = ArrayTwo[count];
}

void BTM(int low, int high){
        int mid;
        if (low < high){
              mid = (low + high) / 2;
              BTM(low,mid);
              BTM(mid+1,high);
              if (my_rank == low){
                    my_pointer[++k] = new unsigned char[ArraySize * i];
                    MPI_Recv(my_pointer[k], ArraySize*i,
                        MPI_UNSIGNED_CHAR,mid+1,0,MPI_COMM_WORLD,&status);
                    my_pointer[++k] = new unsigned char[ArraySize * i* 2];
                    MergeEncryptedData(my_pointer[k-2],my_pointer[k-1],
                                    my_pointer[k], ArraySize * i -1,ArraySize * i -1);
                    i *=2;
              }
              else if(my_rank == mid+1)
                      MPI_Send(my_pointer[k],ArraySize*i,MPI_UNSIGNED_CHAR,
                                        low,0,MPI_COMM_WORLD);
         }
}
```