5-2018

Parallel Implementation of AES using XTS Mode of Operation

Muna Shrestha
St. Cloud State University, shmu1201@stcloudstate.edu

Parallel Implementation of AES Using XTS Mode of Operation

by

Muna Shrestha

A Starred Paper

Submitted to the Graduate Faculty of

St. Cloud State University

in Partial Fulfillment of the Requirements

for the Degree

Master of Science

in Computer Science

May, 2018

Starred Paper Committee:
Andrew A. Anda
Jie H. Meichsner
Mehdi Mekni

Abstract

Data encryption is essential for protecting data from unauthorized access. The Advanced Encryption Standard (AES), among many encryption algorithms, is the most popular algorithm currently employed to secure static and dynamic data. There are several modes of AES operation. Each of these modes defines a unique way to perform data encryption. XTS mode is the latest mode developed to protect data stored in hard-disk-like sector-based storage devices.

A recent increase in the rate of data breaches has triggered the necessity to encrypt stored data as well. AES encryption, however, is a complex process. As it involves a lot of computations, encrypting huge amount of data would undoubtedly be computationally intensive. Parallel computers have been used mostly in high-performance computation research to solve computationally intensive problems. Parallel systems are currently gaining popularity configured as general purpose multi-core system, even at a desktop level. Several programming models have been developed to assist the writing of parallel programs, and some have already been used to parallelize AES. As a result, AES data encryption has become more efficient and applicable.

The message passing model is a popular parallel communication/synchronization model with an early origin. Message Passing Interface (MPI) is the first standardized, vendor-independent, message passing library interface that establishes a portable, efficient, and flexible standard for message passing during computation. Therefore, this paper describes an implementation of AES using XTS mode in parallel via MPI.

Table of Contents

Chapter Page

Chapter Page

List of Tables

List of Figures

Chapter 1: Introduction

Data breaches were in the news headline last year. The Equifax data breach, discovered on July 29, 2017, exploited a website application vulnerability and gained access to certain files. According to the company's' public customer notice of data breach [1], credit card numbers for approximately 209,000 consumers, and certain dispute documents containing personal identifying information for approximately 182,000 consumers, were accessed along with names, social security numbers, birth dates, addresses, and some driver's license numbers. Following that, a print edition, published by The Wall Street Journal on September 18, 2017 [2], stated that more than 11.5 million people had signed up for credit-monitoring offered by Equifax, others had frozen their credit reports with Equifax and its rival companies such as TransUnion and Experian PLC. Moreover, its investors shrunk its stock-market value by about $6 billion, or more than a third, in 10 days, leaving the company crippled. According to a report prepared by Gemalto [3], there were 918 data breaches worldwide during the first half of 2017. It was a 13% increase from the last 6 months of 2016. Equifax, and many other companies, have suffered from such breaches due to the exposure of their data for as little as split second of time. Data security continues to be challenging as more systems, devices, and other objects become interconnected with the growth of the Internet of Things [3].

Less than 5% of all breaches investigated by Gemalto involved encrypted data. Still, many organizations fall short when it comes to data encryption effort. Data breaches take place not only when data is in transit. Data at rest are likely to be breached equally. Exposure of data at rest is equivalent to the exposure of data in transit but for an infinite amount of time. Therefore, as it is undeniable that encryption can actually prevent such breaches, it should be considered as

a necessity rather than just an option. With encryption, data is protected even if infrastructure protections such as firewalls or network access control systems are compromised. Data confidentiality is obtained by converting input plaintext data into ciphertext. With so many potential threats to sensitive data from both inside and outside companies, encrypting is the best way to protect data on personal laptops, storage servers and other devices in addition to emails and communications. Therefore, stored data encryption is the focus of this paper.

Various symmetric algorithms such as AES, Data Encryption Standard (DES), Rivest Cipher 4 (RC4) and Blowfish have been developed to encrypt stored data. These algorithms are similar to each other as all of them use a single key to encrypt data. But they are different in respect to the length of data they process at a time. For example, AES encrypts data in a 128-bit sized block and RC4 encrypts data one bit at a time. However, according to William Stallings [4], AES is comparatively more secure and is currently the most widely used encryption algorithm. As mentioned in Section 3.1, AES can be operated in various modes. XTS is the newest block cipher mode approved by the National Institute of Science and Technology (NIST) in 2010.

With the current trend of recurring data breaches, it is important to encrypt as much data as possible. As encrypting data is itself a complex process, encrypting huge amount of data is also computationally intensive. Solving such computationally intensive problems is always time-consuming. However, advancements in microprocessor technology have yielded more advanced parallel computing systems. Termed parallel computers, these systems have been employed in high-performance computation research to solve computationally intensive problems, and they are now gaining popularity for general purpose computation, even at a desktop level [5].

Using parallel computers for solving most problems is fairly uncommon. The increased availability of parallel computing systems has accelerated interest in exploring the field of parallel programming. The paper, "Parallelizing AES on multicores and GPUs" [6], shows that encrypting data using parallel computers results in speedup factors between 2 and 3. This indicates that data encryption will become more efficient and beneficial if performed in such parallel systems.

Parallel programming is a term used to define the practice of writing a computer program that runs with multiple concurrent execution paths. It is similar to writing a serial program as both involve sequence of steps that tell the computer how to solve a problem. But, unlike serial programming, parallel programming involves multiple processors working together. Therefore, in order to benefit from parallel programming, concurrency within the problem needs to be properly identified and exposed before writing a program.

Section 2.3.1 describes several parallel programming models that have been developed to assist in writing parallel programs. Some of those models have already been used to parallelize AES. However, the decision about which model to use to write a parallel program is usually a combination of a number of factors including the resources available and the nature of the application being developed.

As AES operating in XTS mode can process data blocks independently, it can be implemented in parallel [4]. This being the motivation behind doing this project, a parallel implementation of AES in XTS mode has been performed using message passing model. Parallel programming using MPI is described in detail in Chapter 2. In Chapter 3, AES and its XTS mode are explored, followed by the methodology in Chapter 4 which details on how AES has been

implemented to run on a parallel computer using MPI. In Chapter 5, results obtained from parallel implementation are analyzed and compared against results from its serial implementation. Also, we verify the correctness of the implementation by decrypting its output ciphertexts and comparing the resulting plaintexts with the initial input plaintexts. The source code of both parallel and serial implementations of XTS-AES encryption have been listed as `XTS_AES_Encrypt.c` and `XTS_AES_Serial.c` in Appendix A and Appendix C respectively along with the source code of the parallel implementation of XTS-AES decryption as `XTS_AES_Decrypt.c`. Finally, in Chapter 6, we conclude with our insights.

Chapter 2: Parallel Programming with MPI

Advancements in computer hardware systems, in terms of their memory and processing power, have redefined ways of solving problems using computers. There are newer ways of using computers to make computations more efficient. Programs written using a parallel programming approach are called parallel programs, and they tend to perform better than traditional serial programs. This chapter briefly describes parallel computers first, and then explains how parallel programs are designed and implemented using MPI.

2.1 Parallel Computer

In past, the performance of a computing system had relied on its hardware architecture and had widely been evaluated in terms of its clock speed. Usually expressed in Megahertz (MHz) or Gigahertz (GHz), clock speed basically represents the rate at which a system can complete a processing cycle and depends upon the number of available computing units inside the system. In 1965, Gordon Moore predicted doubling of the number of transistors on microprocessors every 18 to 24 months and following that, computer clock speed doubled roughly every year for more than 40 years. But, in 2005, Gordon Moore himself said his law cannot continue forever because of an increase in heat generation and circuit power requirements [7]. Therefore, performance achievable by increasing clock speed maxed out. Parallel computation was introduced to allow processors to perform better while generating less heat. These parallel computers are commonly available as a multicore processor or group of single processors connected via network or hybrid of both.

Processors in a parallel computer generally execute the same or different program instruction simultaneously on the same or different data. Unlike a serial computer, this type of

machine provides multiple execution paths for instructions. Each processor can have its own local memory or share same memory resources or a mix of both. Among multiple ways of classifying parallel computers, *Flynn's Taxonomy* is the most popular. According to M. J. Flynn [8], such computers are characterized based upon the maximum number of simultaneous instructions or data being in the same phase of execution. Single Instruction Single Data (SISD), Multiple Instruction Single Data (MIMD), Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD) are Flynn's four classification categories, and a parallel computer can belong to any of them.

2.2 Design of a Parallel Program

An algorithm is a set of well-defined steps that lead to the solution of a problem in finite time. Algorithms are essential for solving problems using computers. As in serial programming, parallel programming includes an algorithm design phase that generates a parallel algorithm. A parallel algorithm defines steps to solve a problem using multiple processors. Therefore, in addition to just specifying steps, designing a parallel algorithm requires identifying and exposing concurrency within the problem. Ian Foster [9], has presented a methodological approach for creating a parallel algorithm and it has four distinct stages:

2.2.1 Stage 1–Partitioning

In this stage, the main problem is divided into smaller computation units to expose opportunities for its parallel execution. Such smaller units are called tasks and are created using either domain decomposition or functional decomposition [9].

2.2.1.1 **Domain Decomposition:** In this approach, data associated with the problem is divided into small partitions of nearly equal size. This decomposed data may be an input to the program, an output of the program, or intermediate values maintained by the program.

2.2.1.2 **Functional Decomposition:** In this alternative approach, the computation to be performed is decomposed before decomposing data used by it. After successfully dividing the computation into disjoint tasks, data requirements of tasks are examined. In case of disjoint data requirements, partitioning is complete, otherwise if data overlap is present, significant communication is required to avoid less desirable data replication.

2.2.2 Stage 2–Communication

Tasks created in an initial partitioning stage are expected to execute concurrently and independently from each other. However, computations performed by a task may require data associated with another task. In this case, communication between tasks is necessary to proceed the computation further. In this stage, such communication requirements are identified and respective communication operations are introduced to satisfy all requirements. In general, such communication patterns are categorized along four loosely orthogonal axes: local/global, structured/unstructured, static/dynamic and synchronous/asynchronous [9].

In local communication, a small set of tasks communicate with each other. Each task communicates with few other tasks only. In contrast, global communication takes place between many tasks. Each task communicates with a comparatively larger number of other tasks.

In structured communication, a group of communicating tasks forms regular structure, such as a tree or grid. This structure does not change over time. Whereas, unstructured communication networks could be an arbitrary graph whose structure might change over time.

In static communication, identities of communicating tasks do not change over time. In contrast, the identity of each task may be determined at runtime in dynamic communication, making it highly variable.

In synchronous communication, tasks execute in a coordinated fashion. Communicating tasks cooperate in data transfer operations. Whereas, in an asynchronous communication, tasks may randomly obtain data without cooperating.

2.2.3 Stage 3–Agglomeration

The first two stages of the parallel algorithm design process help in identifying a set of tasks and possible communications between them [9]. However, the design needs to be specialized for an efficient execution on any particular parallel computer as there may be more tasks than available processors. Also, the parallel computer itself may not be suitable for efficient execution of small tasks. Therefore, in this stage, tasks identified in partitioning stage are grouped together if possible.

2.2.4 Stage 4–Mapping

In this stage, tasks are mapped to processors such that processors are utilized with less idling and communication costs [9]. Unlike in uniprocessors or shared-memory processors where operating systems and hardware mechanisms are present to map executable tasks to available processors, in other types of processors, general-purpose mapping is done explicitly. Listed below are two basic mapping strategies to minimize total execution time of a parallel program.

- Place concurrently executable tasks on different processors in order to enhance concurrency.

- Place frequently communicating tasks on the same processor in order to enhance locality.

Mapping can be performed statically, before execution, or dynamically at runtime. Tasks created using domain decomposition are easy to map when they are equal sized and require structured local and/or global communication only. In this case, the mapping is subsumed into the agglomeration phase. This results in a Single Program Multiple Data (SPMD) program which has exactly one task per processor and minimum interprocess communication.

In case of domain decomposition resulting in unequal sized tasks and/or unstructured communication patterns, load balancing algorithms are used. These algorithms identify efficient agglomeration and mapping strategies by using heuristic techniques typically. Whereas, for most complex problems in which either total number of tasks or the total amount of computation or communication keeps changing dynamically, dynamic load balancing algorithm is used. This algorithm is mostly executed locally as it is used periodically to determine a new agglomeration and mapping.

Tasks created using functional decomposition usually have shorter execution times and they communicate with other tasks only at the beginning and end of execution. Such tasks are mapped to idle, or almost idle, processors during runtime using task-scheduling algorithms.

2.3 Implementation of a Parallel Program

Different programming languages and libraries have been developed to write explicitly parallel programs. Also, numerous programming models are available. Each of these models presents a unique way to write parallel programs using available programming languages and libraries. This section provides more information on such parallel programming models.

2.3.1 Parallel Programming Model

A parallel programming model is an abstraction of a computer system that allows convenient expression of an algorithm and its composition in programs. It is machine architecture-independent and exists above hardware and memory architectures. Several programming models have been developed for explicit parallel programming. Each of these models can be implemented on any underlying hardware architecture [5]. The *shared memory model, threads model, message passing model, data parallel model* and *hybrid model* are some common parallel programming models.

These models define a unique view of an address space that a machine makes available for programming, a degree of synchronization imposed on concurrent activities, and a multiplicity of programs [7]. However, selection of the model to be used is usually a combination of several factors like the resources available and the nature of the problem.

2.3.1.1 **Message Passing Model:** Message passing is one of the oldest and most widely used methodologies for writing parallel programs. It focuses more on solving the problem using multiple processors rather than targeting the hardware being used. Therefore, it has been used since early days of parallel processing [7].

The logical view of a machine supporting this model basically contains *p* processors, each with its own private address space. The data involved is explicitly partitioned before being placed inside address spaces. The processor holding the data and the processor that requires the data cooperate with each other prior to actual data transfer.

Generally, this model supports execution of different programs on each participating processors. Though this results in flexible programs, it makes the job of writing parallel

programs highly unscalable [7]. Therefore most message passing programs are written using SPMD approach which makes most processors execute identical code.

MPI, a message passing library interface specification, primarily addresses message passing model [11]. MPI is a de facto standard defined by a community of parallel computing vendors, computer scientists, and application developers [5].

Before 1992, many parallel computers were based on message passing architectures due to their lower costs compared to shared-address-space architectures [7]. Message passing was a natural way of programming for those types of computers. As a result, a variety of vendor-specific message passing libraries were developed. Most of those libraries suffered from compatibility failures. They were vendor specific and did not run on parallel computers built by other vendors. Furthermore, they were mostly syntactically different from each other. Therefore, the MPI standardization effort began with a workshop on Standards for Message-Passing in Distributed Memory Environment on 1992 [11].

2.3.2 Writing a Parallel Program Using MPI

As mentioned in Section 2.3.1.1, MPI is basically a library interface and therefore has multiple implementations. It contains over 125 routines that are used for parallel processing. Initializing and terminating MPI execution environment, getting information about parallel computing environment, and sending and receiving messages between processors are some commonly performed MPI tasks. It has been implemented in more than one way to meet specific requirements. Among many different types of MPI implementations, Open MPI is an open source implementation that is freely available. Below is a well-known “Hello world” program in parallel, written using the C programming language and Open MPI. Compared to

conventional serial C language implementation of the “Hello world” program, it includes some additional MPI statements that are required for writing the parallel program using MPI.

Program: mpihello.c

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv) {
    int myrank, nprocs;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        printf("Hello from processor %d of %d\n", myrank, nprocs);
    MPI_Finalize();

    return 0;
}
```

Figure 1: Parallel Hello World Program using MPI

Including the MPI header file with `#include <mpi.h>` is the first step in writing any MPI program. This header file contains MPI function declarations. Then, the MPI environment is initialized with `MPI_Init(&argc, &argv)` function. During this initialization, each of MPI's global and internal variables such as a communicator (an object that defines a collection of communicating processes) and unique ranks of all processors, get constructed. After `MPI_Init`, `MPI_Comm_size` and `MPI_COMM_WORLD` are two other main functions that get called.

`MPI_Comm_size` returns the size of a communicator which is a total number of participating processors. `MPI_COMM_WORLD` encloses all available processors and returns the total number of processors requested for the job. Each of these processors in the communicator is assigned an incremental rank starting from zero. These ranks uniquely identify communicating processors while sending and receiving messages. If required, they are also used conditionally to

control an execution of the program. `MPI_Comm_rank` function returns the rank of a processor in the communicator. `MPI_Finalize` is a final routine called to clean up the MPI environment. After this function, no more MPI calls are allowed to be made.

Parallel programs written using MPI are compiled and linked using mpicc. Once the program is compiled successfully, it is executed using mpiexec. Below is an output of the mpihello.c program:

```
Hello from processor 1 of 4
Hello from processor 0 of 4
Hello from processor 3 of 4
Hello from processor 2 of 4
```

As mentioned earlier, this program was compiled as `mpicc mpihello.c –o mpihello` and then executed as `mpiexec –n 4 mpihello`. Four processors are requested to print the “Hello world” statement. Depending upon the number of processors available, these parallel programs can be executed on one or more processors. Given below is a general structure of all MPI programs.

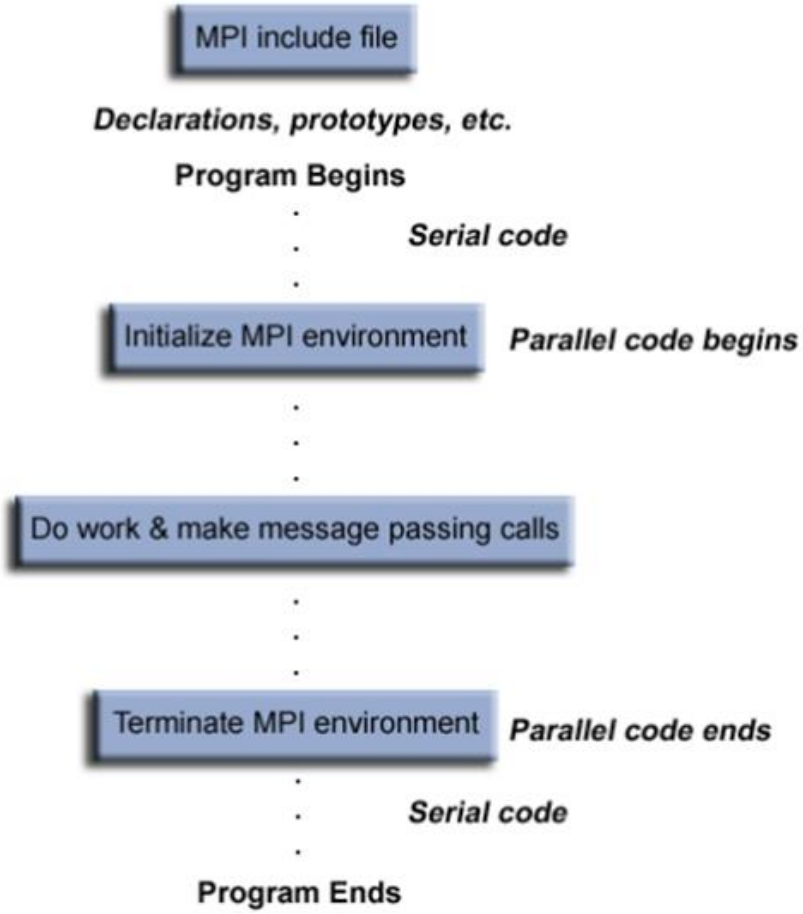

Figure 2: General Structure of a Program Written using MPI [12]

2.4 Communication in MPI Programming

Processors interact by sending and receiving messages in MPI programming [7]. `MPI_Send` and `MPI_Recv` are two functions basically used to send and receive messages between two processors. They are called in the following sequence:

```
int MPI_Send(void *bufSend, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm)
```

```
int MPI_Recv(void *bufRecv, int count, MPI_Datatype
datatype, int source, int tag, MPI_Comm comm, MPI_Status
*status)
```

`MPI_Send` sends data stored in a buffer pointed by pointer `bufSend`. This buffer contains `count` number of consecutive entries of the type specified by parameter `datatype`. Destination of the message sent by `MPI_Send` is uniquely specified by the rank of destination process `dest` and communicator `comm`. Additionally, `MPI_Send` contains an integer-valued `tag` that distinguishes messages from one another.

`MPI_Recv` receives the message sent by a processor whose rank is represented by `source` in `comm`. The tag of the message sent must be that specified by `tag` parameter in the `MPI_Recv` function. In case a processor sends more than one message with an identical tag, only one message is received and stored in continuous locations in a buffer pointed by `bufRecv`. `count` and `datatype` arguments of this function are used to specify the length of the buffer and type of the data it stores. Therefore, the message received should be of type datatype and length equal to or less than count.

In addition to sending and receiving data between just two processors, MPI allows collective communication among the group of participating processors. It provides an extensive set of functions for performing various collective communication operations. *Barrier*, *broadcast*, *scatter*, *gather*, *reduction*, *all-to-all*, and *prefix* are some common types of such operations. Each of these functions contains a communicator as an argument and processors belonging to the communicator communicate by calling functions containing communicator as one of its parameters. Short descriptions of each of these operations follow:

2.4.1 Barrier

This is a synchronization operation performed in MPI using `MPI_Barrier` function. This function call returns only after all processors in the group have called it.

2.4.2 Broadcast

This operation is performed to send data from a processor to all other processors in the group. It is performed in MPI using `MPI_Bcast` function as:

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int source, MPI_Comm comm).
```

2.4.3 Gather

The gather operation is used to collect data from all processors including the one collecting data. It is performed using the `MPI_Gather` function as:

```
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, int target, MPI_Comm comm).
```

2.4.4 Scatter

This operation is used to send a subset of data to all processors including source itself. It is performed using the `MPI_Scatter` function as:

```
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, int source, MPI_Comm comm).
```

2.4.5 All-to-All

During this operation, each processor sends a different portion of the data array to other processors including itself. It is performed using the `MPI_Alltoall` function as:

```
int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype
senddatatype, void *recvbuf, int recvcount, MPI_Datatype
recvdatatype, MPI_Comm comm).
```

2.4.6 Reduce

This operation combines elements stored in the buffer of each processor using an operation specified by `op` and returns combined values. It is performed using the `MPI_Reduce` function as:

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, int target, MPI_Comm comm).
```

2.4.7 Prefix

This operation performs a prefix reduction of data stored in the buffer at each processor. It is performed using the `MPI_Scan` function as:

```
int MPI_Scan(void *sendbuf, void *recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm).
```

Chapter 3: AES and XTS Mode of Operation

Data security is an essential part of today's computer security whether it is securing data in motion or at rest. Block cipher is one of the commonly used encryption algorithms that converts *b*-bit long input block of plaintext into *b*-bit long block of ciphertext using an encryption key. AES is a type of block cipher which was selected by the National Institute of Science and Technology (NIST) as the most secure encryption algorithm in 2001 [9]. This chapter provides details on AES and its operation in XTS mode.

3.1 Background

Encryption is an important technique to secure data. As it protects data even when infrastructure protection such as firewalls or network access control systems get compromised, it is important to be performed using a secure standard like AES. In case of data longer than *b*-bit, it gets broken into multiple *b*-bit long blocks and processed using the same key. According to William Stallings [4], several security issues arise due to this. Therefore, NIST defined different modes of operation to apply a block cipher to a sequence of data blocks or data stream in a variety of applications.

A mode of operation is basically a technique used for enhancing an effect of a cryptographic algorithm or adapting the algorithm itself for an application. Five different modes have been defined for AES data encryption. These modes are Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB) and Counter (CTR). They are summarized in Table 1.

Table 1: Block Cipher Modes of Operation [4]

Mode	Description	Typical Application
Electronic Codebook (ECB)	Each block of plaintext bits is encoded independently using the same key.	• Secure transmission of single values (e.g., an encryption key)
Cipher Block Chaining (CBC)	The input to the encryption algorithm is the XOR of the next block of plaintext and the preceding block of ciphertext.	• General-purpose block-oriented transmission • Authentication
Cipher Feedback (CFB)	Input is processed *s* bits at a time. Preceding ciphertext is used as input to the encryption algorithm to produce pseudorandom output, which is XORed with plaintext to produce next unit of ciphertext.	• General-purpose stream-oriented transmission • Authentication
Output Feedback (OFB)	Similar to CFB, except that the input to the encryption algorithm is the preceding encryption output, and full blocks are used.	• Stream-oriented transmission over noisy channel (e.g., satellite communication)
Counter (CTR)	Each block of plaintext is XORed with an encrypted counter. The counter is incremented for each subsequent block.	• General-purpose block-oriented transmission • Useful for high-speed requirements

XTS is a newer mode introduced by IEEE Security in Storage Working Group (SISWG) to encrypt data stored in hard-disk-like sector-based storage devices [9].

3.2 XTS Mode of Operation

XTS is a XEX encryption mode with tweak and ciphertext stealing feature [9]. It is a tweakable mode developed as an alternative to LRW mode proposed by SISWG as the most promising mode in its early drafts. Unlike CBC mode that has been in use for many years, XTS has complete support for parallelism in its structure. A block cipher in this mode consists of three inputs: a plaintext (*P*) to be encrypted, a symmetric key (*K*), and a tweak (*T*). An output from the encryption (E) is a ciphertext (*C*) represented as E (*K, T, P*). Likewise, decryption (D) of a ciphertext (*C*) results in plaintext (*P*) represented as D (*K, T, C*). Here, in both encryption and

decryption, while key provides security, tweak provides variability, i.e., encrypting a plaintext using the same key but different tweaks result in different ciphertexts and decrypting a ciphertext using the same key but different tweaks result in different plaintexts.

3.2.1 AES using XTS Mode of Operation (XTS-AES)

Figure 3(a) shows the encryption and Figure 3(b) shows the decryption of a single 128-bit block of input plaintext using XTS-AES.

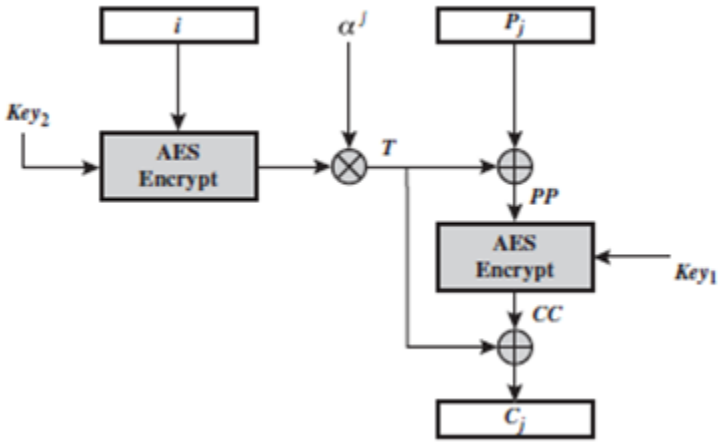

Figure 3(a): XTS-AES Encryption on Single Block [4]

This XTS-AES single block encryption can be represented by following equations as well:

$$C_j \leftarrow \text{XTS-AES-blockEnc}(Key, P_j, i, j)$$

$$T = \text{E}(Key2, i) \otimes \alpha^j$$

$$PP = P_j \oplus T$$

$$CC = \text{E}(Key1, PP)$$

$$C_j = CC \oplus T$$

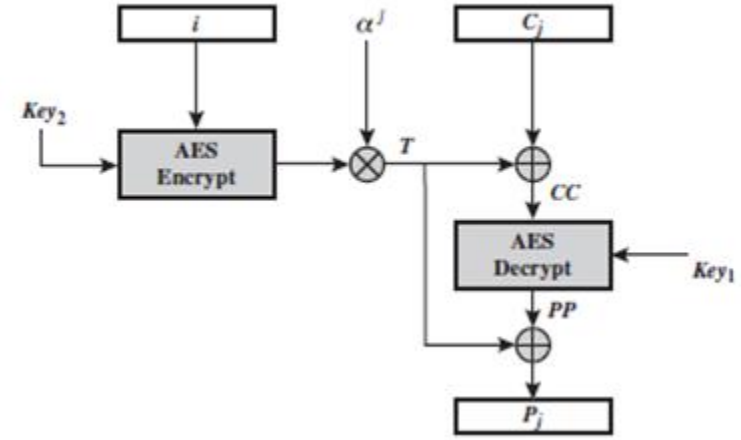

Figure 3(b): XTS-AES Decryption on Single Block [4]

This XTS-AES single block decryption can be represented by following equations as well:

$$P_j \leftarrow \text{XTS-AES-blockDec}(Key, C_j, i, j)$$

$$T = \text{E}(Key2, i) \otimes \alpha^j$$

$$CC = C_j \oplus T$$

$$PP = \text{D}(Key1, CC)$$

$$P_j = PP \oplus T$$

where:

XTS-AES-blockEnc = Single block encryption.

XTS-AES-blockDec = Single block decryption.

Key = The 256 or 512-bit XTS-AES key, parsed as a concatenation of two fields of equal size called *Key1* and *Key2*, such that *Key* = *Key1* || *Key2*.

P_j = The j^{th} block of 128-bit of plaintext.

C_j = The j^{th} block of 128-bit of ciphertext.

i = The 128-bit nonnegative integer tweak value.

j = The sequential number of the 128-bit block inside the data unit.

α = A primitive element of Galois Field (GF (2^{128})) that corresponds to polynomial x.

α^j = α multiplied by itself j times in GF (2^{128}).

$\otimes$ = Multiplication in GF (2^{128}) i.e. modulo multiplication of two polynomials with binary coefficients modulo $x^{128}+x^7+x^2+x+1$.

$\oplus$ = Bitwise XOR.

As shown in Figure 3(a) and Figure 3(b), both single block encryption and single block decryption begin with encryption of 128-bit tweak using AES encryption algorithm and a symmetric key, *Key2*. The resulting output is multiplied by the *j*-th power of α (α^j) in Galois Field to produce *T*. Now for encryption, input plaintext *P* is XORed with *T* first. Then it is processed using AES encryption algorithm and another symmetric key *Key1*. An output of this encryption is again XORed with *T* to produce the final ciphertext *C*. Likewise, in case of decryption, the ciphertext *C* is XORed with *T* before processing with AES decryption algorithm and key *Key1*. Then, an output from the decryption is XORed with *T* again to produce the final plaintext *P*.

In case of data longer than 128-bit, it is divided into *m* number of 128-bit sized blocks. These blocks are then processed independently from each other as shown in Figure 4. This important ability of XTS to process blocks independently supports parallelism in its structure. It induces concurrency and allows a large amount of data to be processed with comparatively more efficiency. Therefore, it is the primary motivation behind implementing XTS-AES in parallel.

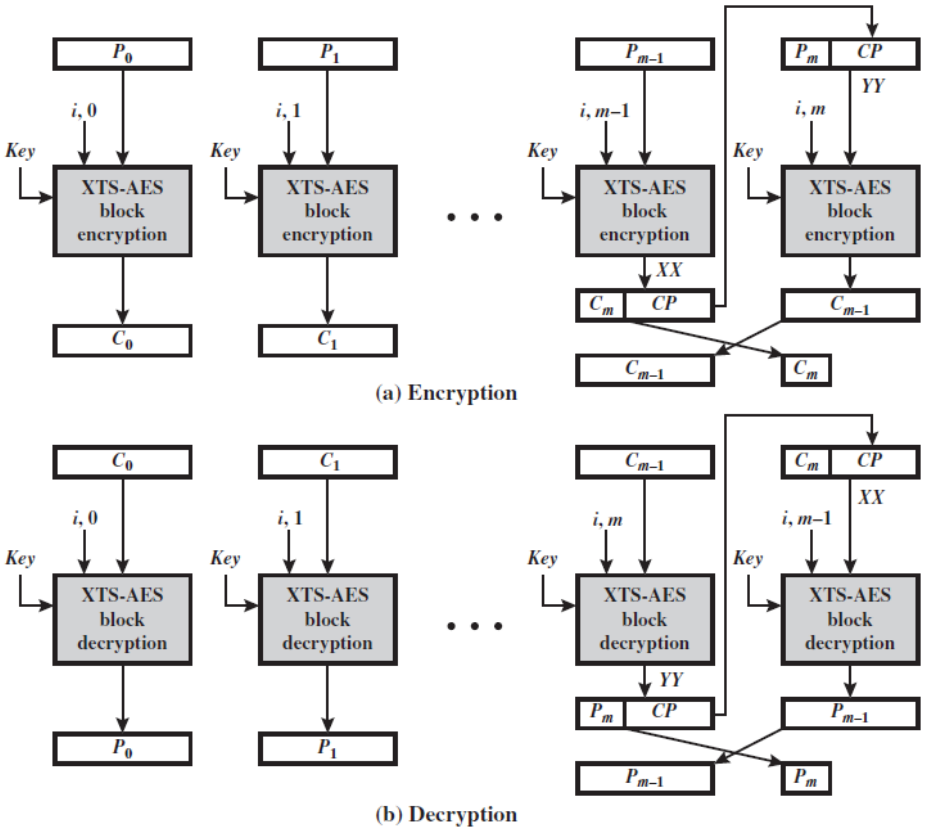

Figure 4: XTS-AES Mode with Ciphertext Stealing [4]

3.2.2 Ciphertext Stealing

As mentioned above, each 128-bit block, P_0 to P_m is encrypted independently to produce ciphertexts C_0 to C_m respectively. However, the last m^{th} block may be a partial block, i.e., it may be 1 to 127 bits long. In such case, a ciphertext-stealing technique is used as shown in Table 2. It is carried out by processing last two blocks in a different way. Initial m-2 blocks are encrypted with XTS-AES single block encryption as XTS-AES-blockEnc(K, P_j, i, j). Also, m-1th block is encrypted similarly and its output is stored in XX. Most significant s-bit of XX are used as s-bit C_m and least significant 128-s bits of XX are concatenated with last input block P_m. Finally, an output of this concatenation, YY is encrypted with $j = m$ to produce ciphertext C_{m-1}.

This ciphertext stealing technique is also applied to recover plaintext P_m from s-bit ciphertext C_m. For that, C_0 to C_{m-2} are decrypted using a single block XTS-AES decryption (XTS-AES-blockDec(K, C_j, i, j)) initially. Likewise, m-1th ciphertext is decrypted to produce YY. Then most significant s-bit of YY are used as s-bit P_m and least significant 128-s bits of YY are concatenated with last ciphertext block C_m. Finally, an output of this concatenation, XX, is decrypted with $j = m$ to produce plaintext P_{m-1}.

Table 2: XTS-AES Mode [4]

XTS-AES mode with null final block	$C_j = \text{XTS-AES-blockEnc}(K, P_j, i, j) \quad j = 0, \ldots, m-1$
	$P_j = \text{XTS-AES-blockEnc}(K, C_j, i, j) \quad j = 0, \ldots, m-1$
XTS-AES mode with final block containing s bits	$C_j = \text{XTS-AES-blockEnc}(K, P_j, i, j)\ j = 0, \ldots, m-2$ $XX = \text{XTS-AES-blockEnc}(K, P_{m-1}, i, m-1)$ $CP = \text{LSB}_{128-s}(XX)$ $YY = P_m \Vert CP$ $C_{m-1} = \text{XTS-AES-blockEnc}(K, YY, i, m)$ $C_m = \text{MSB}_s(XX)$
	$P_j = \text{XTS-AES-blockDec}(K, C_j, i, j)\ j = 0, \ldots, m-2$ $YY = \text{XTS-AES-blockDec}(K, C_{m-1}, i, m-1)$ $CP = \text{LSB}_{128-s}(YY)$ $XX = C_m \Vert \text{CP}$ $P_{m-1} = \text{XTS-AES-blockDec}(K, XX, i, m)$ $P_m = \text{MSB}_s(YY)$

3.3 Advanced Encryption Standard (AES)

As mentioned in Section 3.2, AES algorithm intakes 128-bit block of data to perform encryption or decryption and a key in both cases. Initially, a 128-bit block of input plaintext is represented as a 4 × 4 square matrix of bytes. Then it is copied into a state array and processed as a single matrix on each processing round. After last round, the state array is copied into an output matrix.

AES uses either 16, or 24, or 32-byte (128, 192, or 256-bit) of a symmetric key. Depending upon the length of the key, it is also referred to as AES-128, AES-192, or AES-256 and it contains 10, 12, or 14 processing rounds respectively. Except for the final N^{th} round, all other N-1 rounds consist of four different stages to perform bytes substitution, shift rows, mix columns and add round keys. The final round doesn't consist mix columns stage whereas there is just add round key stage in Round 0 before the first round. Figure 5 shows AES encryption and decryption in N rounds including Round 0.

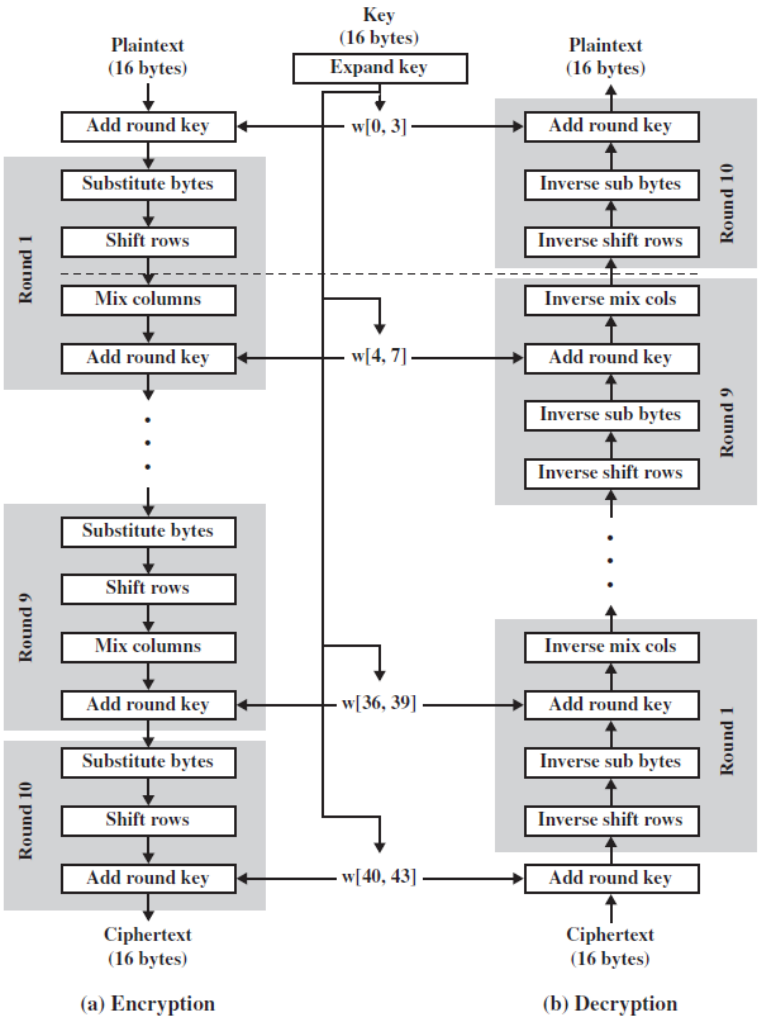

Figure 5: AES Encryption and Decryption [4]

3.3.1 Substitute Bytes

In this stage, a byte-by-byte substitution of input data block is performed using a 16×16 matrix of bytes called an S-box (Table 3). This S-box contains permutations of all possible 256 8-bit values that will be used to substitute each individual byte of input state matrix. As shown in Figure 6, the leftmost nibble of an input byte is used as a row value (x) and the rightmost nibble is used as a column value (y). Both x and y serve as indexes in the S-box to select a unique 8-bit output value. Similarly, an inverse S-box (Table 4) is used in AES decryption during inverse substitute bytes stage.

Table 3: AES S-box [4]

		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	D1	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

Table 4: AES Inverse S-box [4]

		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	4C	95	0B	42	FA	C3	4E
	3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	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	F1	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	E1	69	14	63	55	21	0C	7D

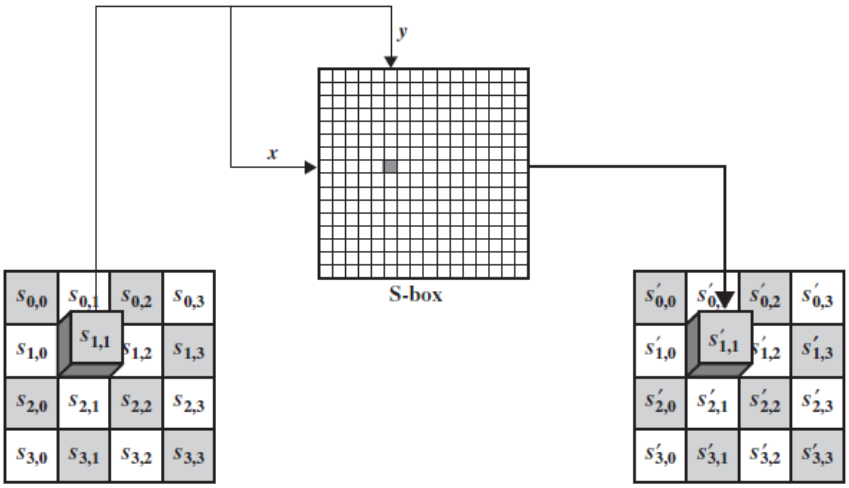

Figure 6: Substitute Byte Transformation [4]

3.3.2 Shift Rows

As shown in Figure 7, for encryption, a simple permutation is carried out by performing a 1-byte circular left shift of each row starting at second row on this stage. Similarly, a 1-byte circular right shift is performed in case of inverse shift rows for decryption.

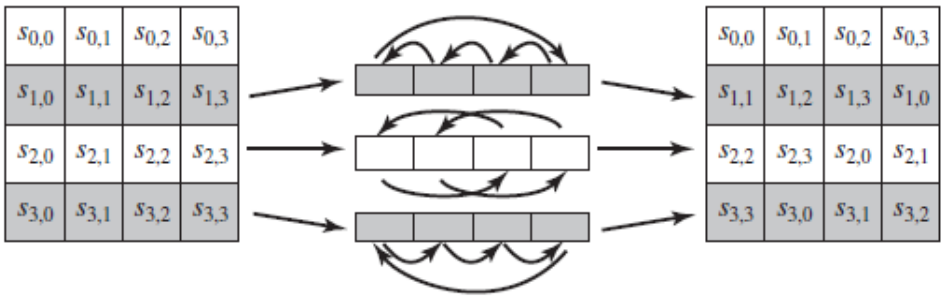

Figure 7: Shift Row Transformation [4]

3.3.3 Mix Columns

In this stage, each byte of a column in the state matrix is mapped into a new value by matrix multiplication with the state matrix as defined in Figure 8. Similarly, the inverse mix column is defined by the matrix multiplication in Figure 9. Each element in the resulting matrix is a sum of products of elements of one row and one column. Here, all additions and multiplications are defined over GF (2^8).

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix}$$

Figure 8: Mix Column Transformation [4]

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix}$$

Figure 9: Inverse Mix Column Transformation [4]

3.3.4 Add Round Key

In this stage, a simple bitwise XOR is performed column-wise between 128-bit of the input state matrix and 128-bit of a round key for both encryption and decryption. 128-bit round keys used by N+1 rounds including Round 0 are produced using an AES key expansion algorithm. As each of these round keys is generated in a different way, knowledge of a part of the cipher key or round key does not enable calculation of many other round keys. Therefore, this stage ensures the security of AES algorithm in general.

3.4 Key Expansion

Depending upon the type of the AES, i.e., AES-128/AES-192/AES-256, a 16/24/32-byte input key is expanded into a linear array of 44/52/60 words following pseudo code in Figure 10. The AES takes the input key (key) and expands key to generate a total of N_b (N_r + 1) words. Here, N_b is the number of words each round requires, and its value is 4 usually. N_r represents a total number of processing rounds for which the key schedule should be prepared. N_k is a total number of words initially available. As shown in Table 5, values of both N_r and N_k depend upon the type of AES used. Additionally, this algorithm requires an initial set of N_k words, and each round requires N_b words long key. The resulting key schedule consists of a linear array of 4-byte words, w[i], with i in a range from 0 to $N_b(N_r + 1)$.

Table 5: Values of N_r and N_k

	N_r	N_k
AES-128	10	4
AES-192	12	6
AES-256	14	8

```
begin
    word temp
    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+1
    end while

    i = Nk

    while (i < Nb * (Nr+1)]
        temp = w[i-1]
        if (i mod Nk = 0)
            temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
        else if (Nk > 6 and i mod Nk = 4)
            temp = SubWord(temp)
        end if
        w[i] = w[i-Nk] xor temp
        i = i + 1
    end while

end
```

Figure 10: Key Expansion Algorithm [11]

Expansion of the input key proceeds according to the pseudo code mentioned above. It can be seen that initial N_k words of an expanded key are filled by dividing the input key into N_k words. After that, every following word, w[*i*], is generated by XORing previous word, w[*i*-1]

with N_k positions earlier word, w[i-N_k]. For words in positions that are multiple of N_k, a transformation is applied to w[i-1]. This transformation consists of a cyclic shift of bytes in w[i-1] followed by bytes substitution on each byte of its input word using the S-box in Table 3. Then its output is XORed with a round constant in a round constant word array Rcon[]. Rcon[] contains values given by [x_{i-1},{00},{00},{00}] with x_{i-1} being powers of x ({02}) in the field GF(2^8).

3.5 Multiplication in Galois Field (GF(2^n))

A field is defined as a set of elements in which addition, subtraction, multiplication, and division can be performed without leaving the set. Rational numbers, real numbers, and complex numbers are some examples of fields. Finite fields and infinite fields are basically two types of field. Finite fields play a crucial role in the context of cryptography. The Galois field (GF (2^n)) is a finite field that consists of 2^n elements and is used in XTS-AES for multiplication.

According to William Stallings [4], there is no simple XOR operation that will accomplish multiplication in GF(2^n). But it also provides a readily generalizable technique with reference to GF (2^8) using irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$.

In GF (2^8), when a polynomial $f(x)$ represented as $b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$, is multiplied by x, we get $x * f(x) = (b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x)$ mod $m(x)$. If b_7 is 0 then its product is a polynomial of degree less than 8 and is already in a reduced form. In this case no further computation is required. But if b_7 is 1, then the product is achieved as: $x * f(x) = (b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + b_0x) + (x^4 + x^3 + x + 1)$. This follows that multiplication by x (i.e., 00000010) can be implemented as a 1-bit left shift operation followed by a conditional bitwise XOR with 00011011 representing $x^4 + x^3 + x + 1$.

In summary for GF(2^8), $x * f(x) = (b_6b_5b_4b_3b_2b_1b_00)$ if $b_7 = 0$. Else if $b_7 = 1$, $x * f(x) = (b_6b_5b_4b_3b_2b_1b_00)$ XOR (00011011) and multiplication by a higher power of x can be achieved by its repeated application.

Chapter 4: Methodology

Parallel implementation of AES using XTS mode of operation is a C implementation of an XTS-AES algorithm which can be executed simultaneously in more than one processor. It has been implemented to optimize encryption of data stored in hard-disk like storage devices. For that, a parallel XTS-AES encryption algorithm is designed following the methodological approach described in Chapter 2 and then implemented in C using message passing parallel programming model with MPI. This chapter provides details on how both design and implementation of the parallel XTS-AES algorithm have been carried out in this project.

4.1 Design of Parallel XTS-AES Algorithm

As mentioned in Chapter 1, encryption of stored data lies at the center of this project. However, in addition to specifying steps to encrypt data, designing a parallel XTS-AES algorithm includes an additional responsibility of identifying and exposing concurrency. As a result, creating a parallel XTS-AES algorithm that runs in *p* number of processors involves following steps:

Step 1–Partitioning Problem into Concurrent Tasks: In this step, task of encrypting high amount of data is partitioned into smaller tasks by using data decomposition. Input data is partitioned into 128-bit long blocks to be encrypted concurrently. Therefore, data blocks represent most tasks in the parallel XTS-AES algorithm. Additionally, a functional decomposition is also performed to create few functional tasks such as preprocessing data, key expansion, encryption, ciphertext stealing, and handling output.

Step 2–Managing Communication between Interacting Tasks: The parallel XTS-AES algorithm is designed to encrypt each 128-bit data block independently. Distributing data among participating processors involves sending data from a root processor (a processor with rank 0) to all participating processors including itself. Also functional tasks may need data being processed by another task. However, the identity of these communicating tasks would stay same throughout the encryption, and data transfer would take place only at the beginning, while distributing input data among processors, and at the end while gathering outputs from them. Therefore, communication is also static and synchronous.

Step 3–Agglomerating Tasks Created from Partitioning: As the size of the data set being processed would always be large, the total number of tasks created by partitioning the data would also be greater than the total number of processors available. Tasks resulting from functional decomposition are also performed on all data. Therefore, the design is customized to create similar *p* coarse-grained tasks, one per processor. Each task contains a certain number of data blocks and pre-defined functional tasks.

Step 4–Mapping Concurrent Tasks onto Processors: As identical tasks would be created after an agglomeration step and required communication between them is static, global, and synchronous, mapping tasks to processors is subsumed in the agglomeration stage. Each processor is assigned exactly one task by using a block distribution of the input array.

In general, like in most message passing programs, the SPMD approach has been used in this project to expose concurrency. Basically, in the SPMD approach, all processors have exact same copy of the program, but they execute different part of it. For example in Figure 11,

`a.out` is an output generated by compiling a parallel program (`a.c`). All tasks mapped to different processors have `a.out` and each of them can execute its different parts concurrently.

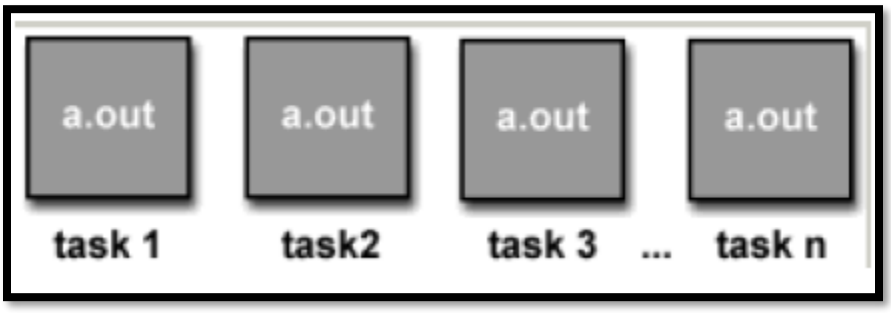

Figure 11: SPMD Program Execution [5]

`XTS_AES_Encrypt.c` in Appendix A is the parallel message passing XTS-AES program that gets compiled into `XTS_AES_Encrypt.out`. It is then executed by all participating processors. It follows the general structure of a parallel program (Figure 2) as well. Figure 12 shows a flowchart that demonstrates how `XTS_AES_Encrypt.c` has been implemented in this project.

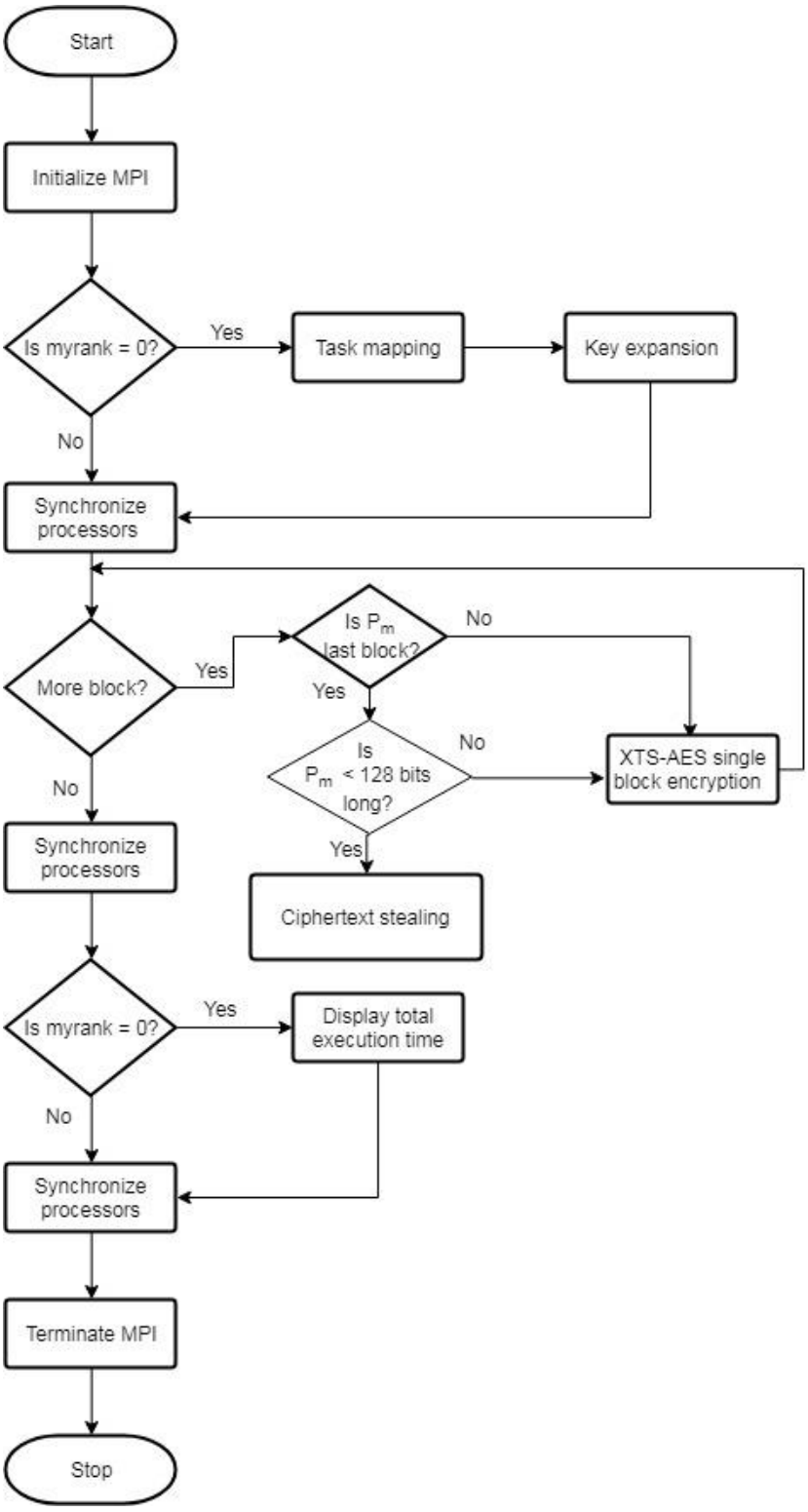

Figure 12(a): Flowchart of Parallel XTS-AES Encryption Algorithm

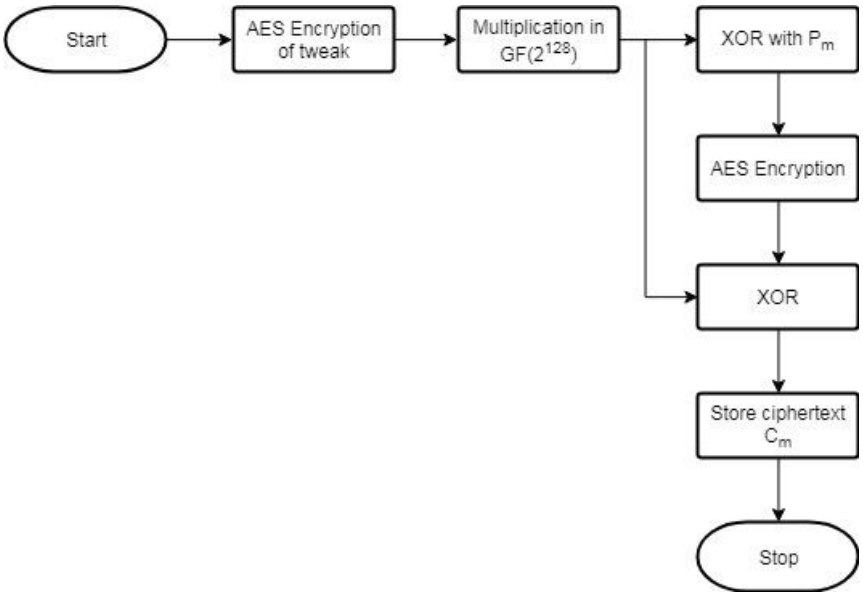

Figure 12(b): Flowchart of XTS-AES Single Block Encryption

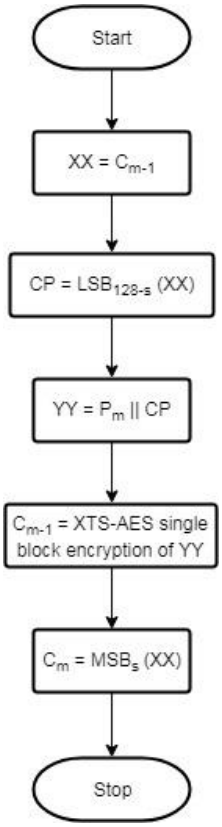

Figure 12(c): Flowchart of Ciphertext Stealing

4.2 Implementation of Parallel XTS-AES Algorithm

As mentioned earlier, `XTS_AES_Encrypt.c` has been implemented in C using MPI. Below is an implementation detail of some important operations performed in parallel XTS-AES.

4.2.1 Task Mapping

A task in parallel XTS-AES is basically a partition of data that needs to be encrypted and operations performed on it. Mapping tasks to processors generally means distributing data partitions among participating processors. Practically, XTS-AES is designed to process data stored in sector-based devices. But in this project data have been stored in a text file `input.txt` in hexadecimal format. As all digits in hexadecimal number system (0, 1,….,9, a, …,f) can be represented in binary using 4-bit, each character from `input.txt` file is considered to be just 4-bit long. These characters are read from the input file into an unsigned character array `global_input` one byte at a time. Size of the `global_input` array is determined at run time from a predefined *TOTAL_NUMBER_OF_ITEMS* constant. Total number of characters in input.txt file is manually stored in *TOTAL_NUMBER_OF_ITEMS* before compiling the program.

Usually the root processor, is assigned task of reading data from a file to an array. Additionally, this processor also performs a block distribution of the array using `MPI_Scatter` function as `MPI_Scatter(global_input, numi, MPI_UNSIGNED_CHAR, &local_input, numi, MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD)`. An input parameter *numi* in this function represents size of each partition, i.e. total number of 8-bit elements of type `MPI_UNSIGNED_CHAR` in the partition. Its value is also set manually as a

result of *TOTAL_NUMBER_OF_ITEMS* divided by (2 * number of processors being used) if a resulting remainder is zero. In case the remainder is not zero, *numi* is (*TOTAL_NUMBER_OF_ITEMS* mod (2 * number of processors being used)) + 1. `local_input` array is another unsigned character array of size *numi* and it is used by each processor to store data partition received from the root processor.

A text file "tweak.txt" has been used to store tweaks that represent nonnegative 128-bit integer values, assigned consecutively to each data unit. Like input data, these tweaks are also read from tweak.txt file into an unsigned character array `global_tweak` by the root processor. As number of tweaks equals to number of input data +/- *s* where 1 < *s* < 128, size of this array is equal to size of the `global_input` array. It is also distributed among same processors by the root processor using `MPI_Scatter` function as *`MPI_Scatter(global_tweak, numi, MPI_UNSIGNED_CHAR, &local_tweak, numi, MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD)`*. Here, `local_tweak` array imitates `local_input` array but stores tweaks instead.

4.2.2 Key Expansion

Key expansion is an important operation performed by the root processor. Generally, XTS-AES has either a 256 or 512-bit key which is parsed as a concatenation of two equal sized fields, *key1* and *key2* such as *Key = key1 || key2*. But in this project, *key1* and *key2* are used directly instead. They are initialized as unsigned character arrays `key1` and `key2` of size either 24-bit or 32-bit. *key1* is used to encrypt input data and *key2* is used to encrypt tweaks. The KeyExpansion function defined in Section 3.4 is called twice to generate round keys that are used in every round of these encryptions. `keysize` is one of the input parameters of this

function. Its value is passed directly while calling the function. Depending upon this value, values of other local variables within the function are set. An output key schedule is stored in an unsigned integer array `w` passed to the function as another input parameter. This array `w` is filled using an expansion algorithm as described in Section 3.4. Round constants used in the algorithm are stored in an unsigned integer array `Rcon` initially. Function RotWord takes a word as an input and performs concatenation of results obtained by shifting the input word to left by 8-bit and shifting it right by 24-bit. SubWord is another function used for substituting the input word byte-wise. Each byte is looked up in the S-Box (Table 3) and replaced by its corresponding value as explained in Section 3.3.1.

4.2.3 Input Data Processing

Once the root processor completes distribution of data among processors as well as expansion of key, each processor starts encrypting data. Both input data and tweaks are copied from `local_input` array and `local_tweak` array into array `plaintexts` and array `tweaks` respectively. Both of these are two dimensional unsigned character arrays that have *numb* numbers of rows and 16 columns. *numb* is a constant representing total number of data blocks within the data partition received by a processor. Its value is also set manually before the program is compiled.

In general, each processor serially encrypts received data using XTS-AES encryption algorithm now. For *numb* times, a processor first reads data from the `plaintexts` array into an unsigned character array `plaintext`. In case `plaintexts` have only *s*-bit such that $0 < s < 128$, then `plaintext[`*s*`+1]` is read from `ciphertexts[`*numb*`-1][`*s*`+1]`. Here, `ciphertexts` array is another two dimensional unsigned character array that stores

corresponding ciphertexts. During this step, whether or not plaintext has 128-bit is recorded. If it is a block of 128-bit input data, then a single block operation is performed, else ciphertext stealing technique is applied. Likewise, tweaks are copied from `tweaks` array to unsigned character array `tweak[16]` before encryption.

4.2.4 Single Block Operation

A single block operation basically consists of two AES encryptions with a multiplication in GF(2^{128}) between them. It performs AES encryption of the input tweak as first AES operation and AES encryption of the input plaintext for second single block AES operation using key schedules obtained from expansion of key *key2* and *key1*, respectively.

A function XTS_AES_Block is called to perform this operation. Its input parameters are `tweak[]`, `plaintext[]`, `key_schedule1`, `key_schedule2` and an index of block being processed currently. It begins with encryption of tweak using aes_encrypt function and obtained result is stored in an unsigned character array `ciphertext_tweak[]`. This is followed by multiplication of data in `ciphertext_tweak[]` with primitive element of GF(2^{128}) multiplied by itself *j* number of times. Here *j* is a sequential number of 128-bit block inside the data unit. Value of *j* is set to (*numb* * rank of the processor) and increased by one for each block of input data. Product of this multiplication is stored in another unsigned character array `T[]` which is used to perform XOR with the plaintext before encrypting it. `T[]` is again used to perform XOR with an output from the encryption of plaintext. In case of ciphertext stealing, i-1th plaintext is readjusted while processing i^{th} plaintext. In this case, P_{m-1} encrypted again.

An output of this second XOR is the final result of single block encryption and it is stored in `ciphertexts[][]` array. Finally, data in `ciphertexts[][]` are gathered at the root processor using `MPI_Gather` function.

4.2.5 AES Encryption

AES_encrypt is a function defined to perform AES encryption. It has four input parameters: two unsigned character array `input[]` and `output[]` that contain its data to be processed and its output respectively, the unsigned integer `key[]` array used for encryption, and an integer variable representing key's size. Encryption in AES is carried out in *N* rounds as described in Section 3.3. Each round contains four steps. For encryption, these steps have been represented by the functions SubBytes, ShiftRows, MixColumns and AddRoundKey. But before initial Round 0, 128-bit input array is copied from `input[]` into 4×4 state square matrix column-wise. This matrix is then updated in each of the *N*+1 rounds, then copied to `output[]`.

4.2.6 Substitute Bytes

Look-up tables with pre-calculated values have been stored in unsigned character arrays `sbox[][]` which is used by a function SubBytes while encrypting. This function substitutes all bytes in the state array with a corresponding value from the look-up table. As this table is represented by arrays, each byte is shifted left 4 times to get first index and then ANDed with 0x0F to get second index.

4.2.7 Shift Rows

In the ShiftRows function, all rows are shifted to left using an integer *t* which stores bytes temporarily to place them in correct position within the state array.

4.2.8 Mix Columns

Each byte in the state array is mapped to a new value by performing column-wise multiplication in GF(2^8), as discussed in Section 3.3.3. The MixColumns function copies data in each column to an unsigned char array `column[4]`. All products have been pre-computed for convenience and stored in the `gf_mul[256][6]` array. The second column of the `gf_mul` array represents multiplicative factors. As total of seven coefficients, 0x01, 0x02, 0x03, 0x09, 0x0b, 0x0d and 0x0e, are only used in the multiplication here, only 6 coefficients are used excluding 1. Therefore, each column of `gf_mul[][]` represents each coefficient mentioned above in an ascending order from 0x00 to 0xFF. Addition involved in this stage is performed using XOR operation.

4.2.9 Add Round Keys

A function AddRoundKey is used for encryption. This function defines an unsigned character array `sub_key[4]` that stores bytes from array `w[]` generated by expanding key. It is updated after performing columnwise XOR with the state array.

4.2.10 Ciphertext Stealing

In case i^{th} input plaintext is only s-bit long, where $0 < s < 128$-bit, its ciphertext is constructed directly using s most significant bits (MSB) of ciphertext of i-1^{th} plaintext and then rest of the 128-s bits are filled with the null character { ‘\0’ }. The i-1^{th} plaintext is also padded with 128-s least significant bits (LSB) of the i-1^{th} ciphertext. Again a single block encryption function XTS_AES_Block is called with input parameters: i-1^{th} tweak and newly created i-1^{th} plaintext, both key schedules and i-1 as the index value.

4.2.11 Multiplication in GF(2^{128})

This is based on technique presented in Section 3.5. Multiplication by *x* is implemented as 1 bit left shift operation followed by a conditional bitwise XOR with 10000111 representing $x^7 + x^2 + x + 1$. As 128-bit data is stored in a 16-byte array, direct XOR operations are not possible in XTS_AES_ Encrypt.c. Therefore, multiplication is performed 1 byte at a time.

The MSB of a byte is stored in an integer variable *MCheckBit*. Then the byte is shifted left by 1 bit. *LCheckBit* is another integer variable initialized to 0. It is used to store MSB of a byte before the one being processed. For example, if byte 2 is being processed currently, *LCheckBit* stores MSB of byte 1.

In case the byte being processed is not the least significant one, depending upon a value in *LCheckBit*, the byte is adjusted. If *LCheckBit* is 1, MSB of the current byte is set to 1 by XORing it with 1. Finally if MSB of the most significant byte is 1, the least significant byte is XORed with 10000111, i.e., 0x87.

4.2.12 Handle Result

After encryption of all data, each processor writes data in `ciphertexts[][]` to `local_input[]` array. `MPI_Gather` function is then called to write those output ciphertexts to `global_input[]`. These data are then written to output.txt file by the root processor. Root processor finally displays total time taken to execute XTS_AES_ Encrypt.c.

4.2.13 Synchronize Processors

To properly synchronize execution of this parallel program, the `MPI_Barrier` function is called multiple times by all processors.

Chapter 5: Result and Analysis

Parallel XTS-AES was implemented to enhance the performance of the XTS-AES algorithm. Various intuitive metrics for evaluating performance of parallel systems exist. An execution time taken to solve a given problem can be considered as the simplest among all [7]. Therefore, execution times of both serial and parallel XTS-AES have been recorded in Table 6 as a function of input size (*n*) and number of processing elements (*p*) used.

Table 6: Execution Time (in milliseconds) for *n* Bytes Data Using *p* Processors

n → / ↓ *p*	1 (Serial)	2	4	8	16	32
256	0.002	0.015	0.02	0.02	0.15	0.1
2560	0.006	0.02	0.02	0.02	0.14	0.04
25600	0.05	0.03	0.03	0.08	0.13	0.1
256000	0.53	0.19	0.17	0.15	0.24	0.17
2560000	5.4	1.5	1.53	1.53	1.68	1.5

In order to evaluate its performance in terms of execution time, parallel XTS-AES has been implemented to run on Intel (R) Xeon (R) CPU ES-2680 v2 @ 2.80 GHz. From data in Table 6, we see that for some data an execution time of the parallel XTS-AES is more than that for serial XTS-AES. This is usually a case when processing smaller sized data.

Unlike in serial algorithms, the execution times of parallel algorithms depend on factors such as the number of processing elements used, and their relative computation and interprocess communication speeds, in addition to their input sizes. A parallel execution time is therefore a sum of computation time with interprocess communication time. As the problem being solved gets divided among participating processors, a ratio of interprocess communication time to actual computation time is comparatively higher for processing smaller-sized data. Therefore, parallel implementations perform better while processing larger amount of data.

As a parallel algorithm cannot be evaluated in isolation from an underlying parallel architecture without some loss in accuracy, some other intuitive metrics, such as speedup, have also been calculated to see how faster XTS_AES_ Encrypt.c run compared to XTS_AES_Serial.c.

Table 7: Speedup of XTS-AES Algorithm for *n* Bytes Data Using *p* Processors

n → / ↓ *p*	2	4	8	16	32
256	0.13	0.1	0.1	0.01	0.02
2560	0.3	0.3	0.3	0.04	0.15
25600	2	2	0.6	0.6	0.6
256000	2.8	3.1	3.5	2.2	3.1
2560000	3.6	3.5	3.5	3.2	3.6

Speedup is defined as a ratio of time taken to solve a problem using a single processing unit to time required to solve the same problem on a parallel computer with *p* identical processing units. Basically, speedup is used to capture a relative benefit of solving a problem in parallel. From the data in Table 7, the average speedup of implemented parallel XTS-AES was calculated to be 1.6. Theoretically, speedup can never exceed the number of processing elements, *p*. In practice, a speedup greater than *p* might be sometimes observed when work performed by a serial algorithm is greater than its parallel formulation, or due to hardware features that put the serial implementation at a disadvantage. This phenomenon, termed super linear speedup, been observed in this project couple of times.

Efficiency is another metric that is used to measure performance of a parallel program. It is a measure of a fraction of time for which a processing element is usefully employed and is calculated as a ratio of speedup to the number of parallel processing elements used. For `XTS_AES_ Encrypt.c`, we found it to be 37%. In the case of an ideal parallel system with *p*

processing elements, a speedup of *p* can be observed. When speedup is equal to *p*, the efficiency is equal to one. However, the inability of parallel processing elements to devote 100% of their time to computation prevents it from being 100% efficient.

In addition to speedup and efficiency, correctness of this program has been verified by decrypting the output ciphertexts. A parallel `XTS_AES_Decrypt.c` (in Appendix B) has been implemented similarly in C using MPI for decryption. Upon decrypting, output plaintexts were found to be identical with input plaintexts in `input.txt` file.

The ability of a parallel program to utilize increasing processing resources effectively is called its *scalability*. It is an important property of any parallel algorithm. It indicates the capability of a program to increase speedup in proportion to the number of processing elements. As performance and correctness of the program are much more difficult to establish based on programs written and tested for smaller problems on fewer processing elements, efficiency has been evaluated under two scenarios:

1. First, for a given problem size, the number of processing elements has been increased. In this case, the efficiency of the parallel program went down as shown in Figure 13. This is a common characteristic of all parallel programs.

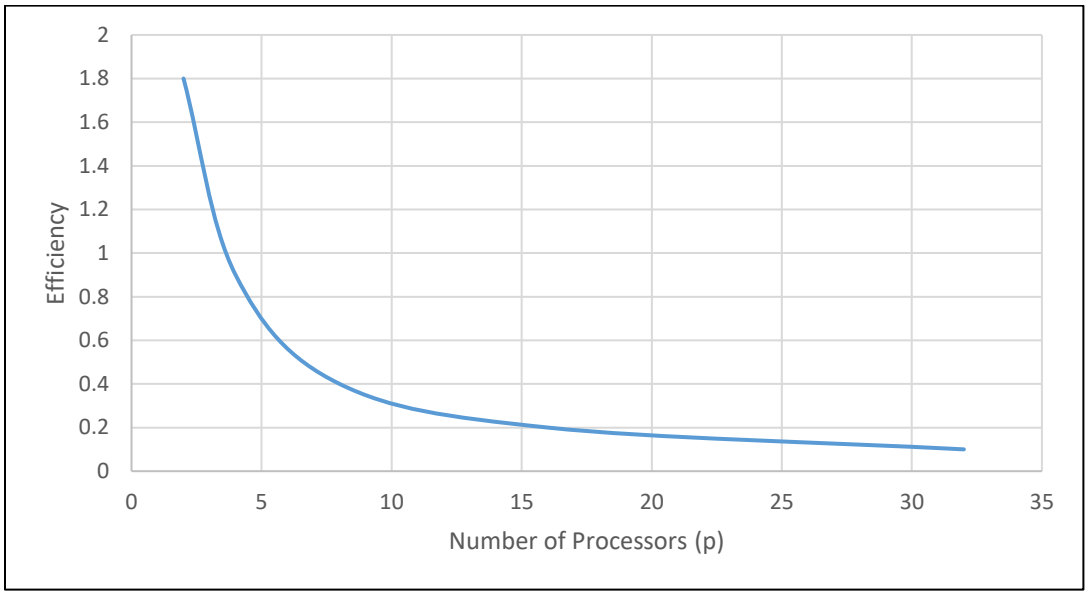

Figure 13: Variation of Efficiency as the Number of Processing Elements is Increased for 256000-byte Data

2. Then, the problem size has been increased, keeping the number of processing elements constant. In this case, efficiency increases with an increase in the problem size, as shown in Figure 14.

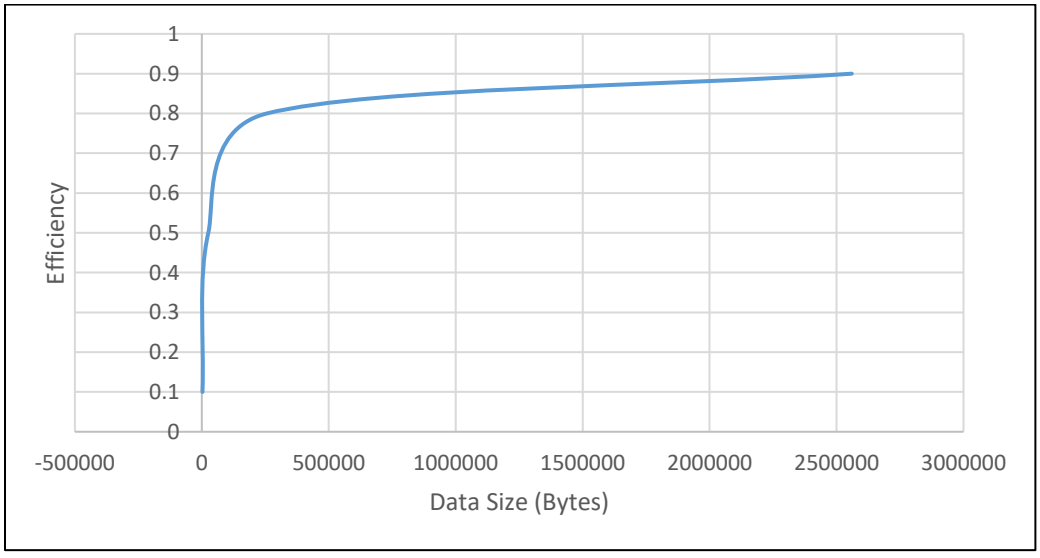

Figure 14: Variation of Efficiency as the Problem Size is Increased for Four Processing Elements

For algorithms exhibiting such characteristics, efficiency could be made more consistent by increasing both the problem size and the number of processing elements simultaneously.

Table 8: Efficiency as a Function of *n* Bytes Input Data on *p* Processing Elements

n → / ↓ *p*	2	4	8	16	32
256	***0.1***	0.1	0.025	0.0125	0.001
2560	0.15	***0.1***	0.04	0.03	0.005
25600	1	0.5	***0.1***	0.03	0.02
256000	1.4	0.8	0.4	***0.1***	0.1
2560000	1.8	0.9	0.4	0.2	***0.1***

In Table 8, the efficiency of encrypting 256-byte data using two processors is 0.1. If the number of processors is increased to 4, and the problem size is increased to 2560-byte, the efficiency remains 0.1. Increasing *p* to 8 and *n* to 25600 results in the same efficiency. Thus, this parallel XTS-AES system can be termed a scalable parallel system since its efficiency remains 0.1 with an increasing number of processing elements, provided that the problem size is also increased.

Chapter 6: Conclusion

Currently, data encryption is gaining popularity within organizations. Increasing trends in disk encryption show that focus has extended from encrypting data-in-motion to data-at-rest. As a result, using more computing units to process larger amounts of data not only increases the rate of data processing but also makes the process comparatively more efficient. XTS is an important mode of AES operation as it provides the flexibility of encrypting data independently. This feature of XTS supports encryption of stored data in parallel.

The message passing model is one of the oldest parallel programming models used widely to write programs that run on parallel computers. MPI, a message passing library interface is a de facto standard for creating portable message-passing parallel programs that can execute across distributed computing nodes. Currently, such programs also run on hybrid systems that contain multi-core CPUs on a single node or connected via a network.

The main objective of this project has been to implement XTS-AES algorithm in parallel. It was preceded by parallel XTS-AES algorithm design. A methodological approach of parallel XTS-AES design has been presented in this paper which provides the theory on how the encryption is carried out by multiple processors. For our implementation, MPI was used to pass messages among processors and to synchronize their actions. Also, the performance of the parallel XTS-AES algorithm was analyzed against its serial implementation. Our results from our analyses show that the implemented parallel XTS-AES algorithm correctly performed encryption with speedup equal to 1.6 and with 37% efficiency. Further, we showed that the parallel XTS-AES algorithm is scalable, as the efficiency remained constant (e.g., 10%) when the number of processing elements and data size both were increased at a constant rate.

Therefore, this implementation can be used to efficiently and reliably encrypt larger amounts of data by using more processors.

References

[1] "Consumer Notice–Cybersecurity Incident & Important Consumer Information | Equifax," n.d., https://www.equifaxsecurity2017.com/consumer-notice/#notice.

[2] A. Andriotis, M. Rapoport, and R. McMillan, "'We've been breached': Inside the Equifax hack" [Editorial], *The Wall Street Journal*, September 18, 2017, https://www.wsj.com/articles/weve-been-breached-inside-the-equifax-hack-1505693318.

[3] "2017 Poor Internal Security Practices Take a Toll" (Rep.)," *Gemalto*, n.d., http://breachlevelindex.com/assets/Breach-Level-Index-Report-H1-2017-Gemalto.pdf.

[4] W. Stallings, *Cryptography and Network Security Principles and Practice*. Boston: Pearson, 2017.

[5] B. Barney, "Introduction to Parallel Computing: Part 1," January 29, 2009, http://www.ddj.com/go-parallel/article/showArticle.jhtml?articleID=212903586.

[6] J. Ortega, H. Trefftz, and C. Trefftz, "Parallelizing AES on multicores and GPUs [Scholarly project]," in *Parallelizing AES on Multicores and GPUs*, August 8, 2011, http://ieeexplore.ieee.org/abstract/document/5978576/authors.

[7] "Fifty Years of Microprocessor Technology Advancements: 1965 to 2015," Itanium Solutions Alliance, n.d.

[8] M. J. Flynn, "Very high-speed computing systems," December 1966, http://ieeexplore.ieee.org/document/1447203/authors.

[9] I. Foster, I. (1995). *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Reading, MA: Addison-Wesley, 1995, http://www.mcs.anl.gov/~itf/dbpp/.

[10] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, Second Edition. Addison-Wesley, 2003.

[11] "MPI: A Message-Passing Interface Standard Version 3.1," June 4, 2015.

[12] B. Barney, n.d., https://computing.llnl.gov/tutorials/mpi/.

[13] M. Alomari and K. Samsudin, "A Parallel XTS Encryption Mode of Operation" [Scholarly project], April 5, 2010, http://ieeexplore.ieee.org/document/5443177/.

[14] Information Technology Laboratory (National Institute of Standards and Technology). "Announcing the Advanced Encryption Standard (AES)," Gaithersburg, MD: Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, 2001.

[15] D. Kinghorn, "5 Ways of Parallel Programming," May 12, 2015, https://www.pugetsystems.com/labs/hpc/5-Ways-of-Parallel-Programming-652/.

Appendix

[A] XTS_AES_Encrypt.c

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

#define TOTAL_NUM_OF_ITEMS 16777216
#define TOTAL_NUM_OF_TWEAKS 16777216
#define numb 32768
#define numi 524288

#define uchar unsigned char // 8 bits
#define uint unsigned long // 32 bits


int idx, LCheckBit, MCheckBit, j=1, jj, k, idx, idxx, first=1;
uchar tweaks[numb][16], tweak[16];
uchar plaintexts[numb][16], plaintext[16], second_last_plaintext[16];
uchar ciphertexts[numb][16], ciphertext[16], ciphertext_tweak[16] ;
uchar T[16], PP[16], CC[16], temp[16];
uint key_schedule1[60],key_schedule2[60];

uchar sbox[16][16] = {
   0x63,0x7C,0x77,0x7B,0xF2,0x6B,0x6F,0xC5,0x30,0x01,0x67,0x2B,0xFE,0xD7,0xAB,0x76,
   0xCA,0x82,0xC9,0x7D,0xFA,0x59,0x47,0xF0,0xAD,0xD4,0xA2,0xAF,0x9C,0xA4,0x72,0xC0,
   0xB7,0xFD,0x93,0x26,0x36,0x3F,0xF7,0xCC,0x34,0xA5,0xE5,0xF1,0x71,0xD8,0x31,0x15,
   0x04,0xC7,0x23,0xC3,0x18,0x96,0x05,0x9A,0x07,0x12,0x80,0xE2,0xEB,0x27,0xB2,0x75,
   0x09,0x83,0x2C,0x1A,0x1B,0x6E,0x5A,0xA0,0x52,0x3B,0xD6,0xB3,0x29,0xE3,0x2F,0x84,
   0x53,0xD1,0x00,0xED,0x20,0xFC,0xB1,0x5B,0x6A,0xCB,0xBE,0x39,0x4A,0x4C,0x58,0xCF,
   0xD0,0xEF,0xAA,0xFB,0x43,0x4D,0x33,0x85,0x45,0xF9,0x02,0x7F,0x50,0x3C,0x9F,0xA8,
   0x51,0xA3,0x40,0x8F,0x92,0x9D,0x38,0xF5,0xBC,0xB6,0xDA,0x21,0x10,0xFF,0xF3,0xD2,
   0xCD,0x0C,0x13,0xEC,0x5F,0x97,0x44,0x17,0xC4,0xA7,0x7E,0x3D,0x64,0x5D,0x19,0x73,
   0x60,0x81,0x4F,0xDC,0x22,0x2A,0x90,0x88,0x46,0xEE,0xB8,0x14,0xDE,0x5E,0x0B,0xDB,
   0xE0,0x32,0x3A,0x0A,0x49,0x06,0x24,0x5C,0xC2,0xD3,0xAC,0x62,0x91,0x95,0xE4,0x79,
   0xE7,0xC8,0x37,0x6D,0x8D,0xD5,0x4E,0xA9,0x6C,0x56,0xF4,0xEA,0x65,0x7A,0xAE,0x08,
   0xBA,0x78,0x25,0x2E,0x1C,0xA6,0xB4,0xC6,0xE8,0xDD,0x74,0x1F,0x4B,0xBD,0x8B,0x8A,
   0x70,0x3E,0xB5,0x66,0x48,0x03,0xF6,0x0E,0x61,0x35,0x57,0xB9,0x86,0xC1,0x1D,0x9E,
   0xE1,0xF8,0x98,0x11,0x69,0xD9,0x8E,0x94,0x9B,0x1E,0x87,0xE9,0xCE,0x55,0x28,0xDF,
   0x8C,0xA1,0x89,0x0D,0xBF,0xE6,0x42,0x68,0x41,0x99,0x2D,0x0F,0xB0,0x54,0xBB,0x16
};
```

```
uchar gf_mul[256][6] = {
    {0x00,0x00,0x00,0x00,0x00,0x00},{0x02,0x03,0x09,0x0b,0x0d,0x0e},
    {0x04,0x06,0x12,0x16,0x1a,0x1c},{0x06,0x05,0x1b,0x1d,0x17,0x12},
    {0x08,0x0c,0x24,0x2c,0x34,0x38},{0x0a,0x0f,0x2d,0x27,0x39,0x36},
    {0x0c,0x0a,0x36,0x3a,0x2e,0x24},{0x0e,0x09,0x3f,0x31,0x23,0x2a},
    {0x10,0x18,0x48,0x58,0x68,0x70},{0x12,0x1b,0x41,0x53,0x65,0x7e},
    {0x14,0x1e,0x5a,0x4e,0x72,0x6c},{0x16,0x1d,0x53,0x45,0x7f,0x62},
    {0x18,0x14,0x6c,0x74,0x5c,0x48},{0x1a,0x17,0x65,0x7f,0x51,0x46},
    {0x1c,0x12,0x7e,0x62,0x46,0x54},{0x1e,0x11,0x77,0x69,0x4b,0x5a},
    {0x20,0x30,0x90,0xb0,0xd0,0xe0},{0x22,0x33,0x99,0xbb,0xdd,0xee},
    {0x24,0x36,0x82,0xa6,0xca,0xfc},{0x26,0x35,0x8b,0xad,0xc7,0xf2},
    {0x28,0x3c,0xb4,0x9c,0xe4,0xd8},{0x2a,0x3f,0xbd,0x97,0xe9,0xd6},
    {0x2c,0x3a,0xa6,0x8a,0xfe,0xc4},{0x2e,0x39,0xaf,0x81,0xf3,0xca},
    {0x30,0x28,0xd8,0xe8,0xb8,0x90},{0x32,0x2b,0xd1,0xe3,0xb5,0x9e},
    {0x34,0x2e,0xca,0xfe,0xa2,0x8c},{0x36,0x2d,0xc3,0xf5,0xaf,0x82},
    {0x38,0x24,0xfc,0xc4,0x8c,0xa8},{0x3a,0x27,0xf5,0xcf,0x81,0xa6},
    {0x3c,0x22,0xee,0xd2,0x96,0xb4},{0x3e,0x21,0xe7,0xd9,0x9b,0xba},
    {0x40,0x60,0x3b,0x7b,0xbb,0xdb},{0x42,0x63,0x32,0x70,0xb6,0xd5},
    {0x44,0x66,0x29,0x6d,0xa1,0xc7},{0x46,0x65,0x20,0x66,0xac,0xc9},
    {0x48,0x6c,0x1f,0x57,0x8f,0xe3},{0x4a,0x6f,0x16,0x5c,0x82,0xed},
    {0x4c,0x6a,0x0d,0x41,0x95,0xff},{0x4e,0x69,0x04,0x4a,0x98,0xf1},
    {0x50,0x78,0x73,0x23,0xd3,0xab},{0x52,0x7b,0x7a,0x28,0xde,0xa5},
    {0x54,0x7e,0x61,0x35,0xc9,0xb7},{0x56,0x7d,0x68,0x3e,0xc4,0xb9},
    {0x58,0x74,0x57,0x0f,0xe7,0x93},{0x5a,0x77,0x5e,0x04,0xea,0x9d},
    {0x5c,0x72,0x45,0x19,0xfd,0x8f},{0x5e,0x71,0x4c,0x12,0xf0,0x81},
    {0x60,0x50,0xab,0xcb,0x6b,0x3b},{0x62,0x53,0xa2,0xc0,0x66,0x35},
    {0x64,0x56,0xb9,0xdd,0x71,0x27},{0x66,0x55,0xb0,0xd6,0x7c,0x29},
    {0x68,0x5c,0x8f,0xe7,0x5f,0x03},{0x6a,0x5f,0x86,0xec,0x52,0x0d},
    {0x6c,0x5a,0x9d,0xf1,0x45,0x1f},{0x6e,0x59,0x94,0xfa,0x48,0x11},
    {0x70,0x48,0xe3,0x93,0x03,0x4b},{0x72,0x4b,0xea,0x98,0x0e,0x45},
    {0x74,0x4e,0xf1,0x85,0x19,0x57},{0x76,0x4d,0xf8,0x8e,0x14,0x59},
    {0x78,0x44,0xc7,0xbf,0x37,0x73},{0x7a,0x47,0xce,0xb4,0x3a,0x7d},
    {0x7c,0x42,0xd5,0xa9,0x2d,0x6f},{0x7e,0x41,0xdc,0xa2,0x20,0x61},
    {0x80,0xc0,0x76,0xf6,0x6d,0xad},{0x82,0xc3,0x7f,0xfd,0x60,0xa3},
    {0x84,0xc6,0x64,0xe0,0x77,0xb1},{0x86,0xc5,0x6d,0xeb,0x7a,0xbf},
    {0x88,0xcc,0x52,0xda,0x59,0x95},{0x8a,0xcf,0x5b,0xd1,0x54,0x9b},
    {0x8c,0xca,0x40,0xcc,0x43,0x89},{0x8e,0xc9,0x49,0xc7,0x4e,0x87},
    {0x90,0xd8,0x3e,0xae,0x05,0xdd},{0x92,0xdb,0x37,0xa5,0x08,0xd3},
    {0x94,0xde,0x2c,0xb8,0x1f,0xc1},{0x96,0xdd,0x25,0xb3,0x12,0xcf},
```

```
{0x98,0xd4,0x1a,0x82,0x31,0xe5},{0x9a,0xd7,0x13,0x89,0x3c,0xeb},
{0x9c,0xd2,0x08,0x94,0x2b,0xf9},{0x9e,0xd1,0x01,0x9f,0x26,0xf7},
{0xa0,0xf0,0xe6,0x46,0xbd,0x4d},{0xa2,0xf3,0xef,0x4d,0xb0,0x43},
{0xa4,0xf6,0xf4,0x50,0xa7,0x51},{0xa6,0xf5,0xfd,0x5b,0xaa,0x5f},
{0xa8,0xfc,0xc2,0x6a,0x89,0x75},{0xaa,0xff,0xcb,0x61,0x84,0x7b},
{0xac,0xfa,0xd0,0x7c,0x93,0x69},{0xae,0xf9,0xd9,0x77,0x9e,0x67},
{0xb0,0xe8,0xae,0x1e,0xd5,0x3d},{0xb2,0xeb,0xa7,0x15,0xd8,0x33},
{0xb4,0xee,0xbc,0x08,0xcf,0x21},{0xb6,0xed,0xb5,0x03,0xc2,0x2f},
{0xb8,0xe4,0x8a,0x32,0xe1,0x05},{0xba,0xe7,0x83,0x39,0xec,0x0b},
{0xbc,0xe2,0x98,0x24,0xfb,0x19},{0xbe,0xe1,0x91,0x2f,0xf6,0x17},
{0xc0,0xa0,0x4d,0x8d,0xd6,0x76},{0xc2,0xa3,0x44,0x86,0xdb,0x78},
{0xc4,0xa6,0x5f,0x9b,0xcc,0x6a},{0xc6,0xa5,0x56,0x90,0xc1,0x64},
{0xc8,0xac,0x69,0xa1,0xe2,0x4e},{0xca,0xaf,0x60,0xaa,0xef,0x40},
{0xcc,0xaa,0x7b,0xb7,0xf8,0x52},{0xce,0xa9,0x72,0xbc,0xf5,0x5c},
{0xd0,0xb8,0x05,0xd5,0xbe,0x06},{0xd2,0xbb,0x0c,0xde,0xb3,0x08},
{0xd4,0xbe,0x17,0xc3,0xa4,0x1a},{0xd6,0xbd,0x1e,0xc8,0xa9,0x14},
{0xd8,0xb4,0x21,0xf9,0x8a,0x3e},{0xda,0xb7,0x28,0xf2,0x87,0x30},
{0xdc,0xb2,0x33,0xef,0x90,0x22},{0xde,0xb1,0x3a,0xe4,0x9d,0x2c},
{0xe0,0x90,0xdd,0x3d,0x06,0x96},{0xe2,0x93,0xd4,0x36,0x0b,0x98},
{0xe4,0x96,0xcf,0x2b,0x1c,0x8a},{0xe6,0x95,0xc6,0x20,0x11,0x84},
{0xe8,0x9c,0xf9,0x11,0x32,0xae},{0xea,0x9f,0xf0,0x1a,0x3f,0xa0},
{0xec,0x9a,0xeb,0x07,0x28,0xb2},{0xee,0x99,0xe2,0x0c,0x25,0xbc},
{0xf0,0x88,0x95,0x65,0x6e,0xe6},{0xf2,0x8b,0x9c,0x6e,0x63,0xe8},
{0xf4,0x8e,0x87,0x73,0x74,0xfa},{0xf6,0x8d,0x8e,0x78,0x79,0xf4},
{0xf8,0x84,0xb1,0x49,0x5a,0xde},{0xfa,0x87,0xb8,0x42,0x57,0xd0},
{0xfc,0x82,0xa3,0x5f,0x40,0xc2},{0xfe,0x81,0xaa,0x54,0x4d,0xcc},
{0x1b,0x9b,0xec,0xf7,0xda,0x41},{0x19,0x98,0xe5,0xfc,0xd7,0x4f},
{0x1f,0x9d,0xfe,0xe1,0xc0,0x5d},{0x1d,0x9e,0xf7,0xea,0xcd,0x53},
{0x13,0x97,0xc8,0xdb,0xee,0x79},{0x11,0x94,0xc1,0xd0,0xe3,0x77},
{0x17,0x91,0xda,0xcd,0xf4,0x65},{0x15,0x92,0xd3,0xc6,0xf9,0x6b},
{0x0b,0x83,0xa4,0xaf,0xb2,0x31},{0x09,0x80,0xad,0xa4,0xbf,0x3f},
{0x0f,0x85,0xb6,0xb9,0xa8,0x2d},{0x0d,0x86,0xbf,0xb2,0xa5,0x23},
{0x03,0x8f,0x80,0x83,0x86,0x09},{0x01,0x8c,0x89,0x88,0x8b,0x07},
{0x07,0x89,0x92,0x95,0x9c,0x15},{0x05,0x8a,0x9b,0x9e,0x91,0x1b},
{0x3b,0xab,0x7c,0x47,0x0a,0xa1},{0x39,0xa8,0x75,0x4c,0x07,0xaf},
{0x3f,0xad,0x6e,0x51,0x10,0xbd},{0x3d,0xae,0x67,0x5a,0x1d,0xb3},
{0x33,0xa7,0x58,0x6b,0x3e,0x99},{0x31,0xa4,0x51,0x60,0x33,0x97},
{0x37,0xa1,0x4a,0x7d,0x24,0x85},{0x35,0xa2,0x43,0x76,0x29,0x8b},
{0x2b,0xb3,0x34,0x1f,0x62,0xd1},{0x29,0xb0,0x3d,0x14,0x6f,0xdf},
```

```
{0x2f,0xb5,0x26,0x09,0x78,0xcd},{0x2d,0xb6,0x2f,0x02,0x75,0xc3},
{0x23,0xbf,0x10,0x33,0x56,0xe9},{0x21,0xbc,0x19,0x38,0x5b,0xe7},
{0x27,0xb9,0x02,0x25,0x4c,0xf5},{0x25,0xba,0x0b,0x2e,0x41,0xfb},
{0x5b,0xfb,0xd7,0x8c,0x61,0x9a},{0x59,0xf8,0xde,0x87,0x6c,0x94},
{0x5f,0xfd,0xc5,0x9a,0x7b,0x86},{0x5d,0xfe,0xcc,0x91,0x76,0x88},
{0x53,0xf7,0xf3,0xa0,0x55,0xa2},{0x51,0xf4,0xfa,0xab,0x58,0xac},
{0x57,0xf1,0xe1,0xb6,0x4f,0xbe},{0x55,0xf2,0xe8,0xbd,0x42,0xb0},
{0x4b,0xe3,0x9f,0xd4,0x09,0xea},{0x49,0xe0,0x96,0xdf,0x04,0xe4},
{0x4f,0xe5,0x8d,0xc2,0x13,0xf6},{0x4d,0xe6,0x84,0xc9,0x1e,0xf8},
{0x43,0xef,0xbb,0xf8,0x3d,0xd2},{0x41,0xec,0xb2,0xf3,0x30,0xdc},
{0x47,0xe9,0xa9,0xee,0x27,0xce},{0x45,0xea,0xa0,0xe5,0x2a,0xc0},
{0x7b,0xcb,0x47,0x3c,0xb1,0x7a},{0x79,0xc8,0x4e,0x37,0xbc,0x74},
{0x7f,0xcd,0x55,0x2a,0xab,0x66},{0x7d,0xce,0x5c,0x21,0xa6,0x68},
{0x73,0xc7,0x63,0x10,0x85,0x42},{0x71,0xc4,0x6a,0x1b,0x88,0x4c},
{0x77,0xc1,0x71,0x06,0x9f,0x5e},{0x75,0xc2,0x78,0x0d,0x92,0x50},
{0x6b,0xd3,0x0f,0x64,0xd9,0x0a},{0x69,0xd0,0x06,0x6f,0xd4,0x04},
{0x6f,0xd5,0x1d,0x72,0xc3,0x16},{0x6d,0xd6,0x14,0x79,0xce,0x18},
{0x63,0xdf,0x2b,0x48,0xed,0x32},{0x61,0xdc,0x22,0x43,0xe0,0x3c},
{0x67,0xd9,0x39,0x5e,0xf7,0x2e},{0x65,0xda,0x30,0x55,0xfa,0x20},
{0x9b,0x5b,0x9a,0x01,0xb7,0xec},{0x99,0x58,0x93,0x0a,0xba,0xe2},
{0x9f,0x5d,0x88,0x17,0xad,0xf0},{0x9d,0x5e,0x81,0x1c,0xa0,0xfe},
{0x93,0x57,0xbe,0x2d,0x83,0xd4},{0x91,0x54,0xb7,0x26,0x8e,0xda},
{0x97,0x51,0xac,0x3b,0x99,0xc8},{0x95,0x52,0xa5,0x30,0x94,0xc6},
{0x8b,0x43,0xd2,0x59,0xdf,0x9c},{0x89,0x40,0xdb,0x52,0xd2,0x92},
{0x8f,0x45,0xc0,0x4f,0xc5,0x80},{0x8d,0x46,0xc9,0x44,0xc8,0x8e},
{0x83,0x4f,0xf6,0x75,0xeb,0xa4},{0x81,0x4c,0xff,0x7e,0xe6,0xaa},
{0x87,0x49,0xe4,0x63,0xf1,0xb8},{0x85,0x4a,0xed,0x68,0xfc,0xb6},
{0xbb,0x6b,0x0a,0xb1,0x67,0x0c},{0xb9,0x68,0x03,0xba,0x6a,0x02},
{0xbf,0x6d,0x18,0xa7,0x7d,0x10},{0xbd,0x6e,0x11,0xac,0x70,0x1e},
{0xb3,0x67,0x2e,0x9d,0x53,0x34},{0xb1,0x64,0x27,0x96,0x5e,0x3a},
{0xb7,0x61,0x3c,0x8b,0x49,0x28},{0xb5,0x62,0x35,0x80,0x44,0x26},
{0xab,0x73,0x42,0xe9,0x0f,0x7c},{0xa9,0x70,0x4b,0xe2,0x02,0x72},
{0xaf,0x75,0x50,0xff,0x15,0x60},{0xad,0x76,0x59,0xf4,0x18,0x6e},
{0xa3,0x7f,0x66,0xc5,0x3b,0x44},{0xa1,0x7c,0x6f,0xce,0x36,0x4a},
{0xa7,0x79,0x74,0xd3,0x21,0x58},{0xa5,0x7a,0x7d,0xd8,0x2c,0x56},
{0xdb,0x3b,0xa1,0x7a,0x0c,0x37},{0xd9,0x38,0xa8,0x71,0x01,0x39},
{0xdf,0x3d,0xb3,0x6c,0x16,0x2b},{0xdd,0x3e,0xba,0x67,0x1b,0x25},
{0xd3,0x37,0x85,0x56,0x38,0x0f},{0xd1,0x34,0x8c,0x5d,0x35,0x01},
{0xd7,0x31,0x97,0x40,0x22,0x13},{0xd5,0x32,0x9e,0x4b,0x2f,0x1d},
```

```
    {0xcb,0x23,0xe9,0x22,0x64,0x47},{0xc9,0x20,0xe0,0x29,0x69,0x49},
    {0xcf,0x25,0xfb,0x34,0x7e,0x5b},{0xcd,0x26,0xf2,0x3f,0x73,0x55},
    {0xc3,0x2f,0xcd,0x0e,0x50,0x7f},{0xc1,0x2c,0xc4,0x05,0x5d,0x71},
    {0xc7,0x29,0xdf,0x18,0x4a,0x63},{0xc5,0x2a,0xd6,0x13,0x47,0x6d},
    {0xfb,0x0b,0x31,0xca,0xdc,0xd7},{0xf9,0x08,0x38,0xc1,0xd1,0xd9},
    {0xff,0x0d,0x23,0xdc,0xc6,0xcb},{0xfd,0x0e,0x2a,0xd7,0xcb,0xc5},
    {0xf3,0x07,0x15,0xe6,0xe8,0xef},{0xf1,0x04,0x1c,0xed,0xe5,0xe1},
    {0xf7,0x01,0x07,0xf0,0xf2,0xf3},{0xf5,0x02,0x0e,0xfb,0xff,0xfd},
    {0xeb,0x13,0x79,0x92,0xb4,0xa7},{0xe9,0x10,0x70,0x99,0xb9,0xa9},
    {0xef,0x15,0x6b,0x84,0xae,0xbb},{0xed,0x16,0x62,0x8f,0xa3,0xb5},
    {0xe3,0x1f,0x5d,0xbe,0x80,0x9f},{0xe1,0x1c,0x54,0xb5,0x8d,0x91},
    {0xe7,0x19,0x4f,0xa8,0x9a,0x83},{0xe5,0x1a,0x46,0xa3,0x97,0x8d}
};

void AddRoundKey(uchar state[][4], uint w[]){
    uchar sub_key[4];

    sub_key[0] = w[0] >> 24;
    sub_key[1] = w[0] >> 16;
    sub_key[2] = w[0] >> 8;
    sub_key[3] = w[0];
    state[0][0] ^= sub_key[0];
    state[1][0] ^= sub_key[1];
    state[2][0] ^= sub_key[2];
    state[3][0] ^= sub_key[3];

    sub_key[0] = w[1] >> 24;
    sub_key[1] = w[1] >> 16;
    sub_key[2] = w[1] >> 8;
    sub_key[3] = w[1];
    state[0][1] ^= sub_key[0];
    state[1][1] ^= sub_key[1];
    state[2][1] ^= sub_key[2];
    state[3][1] ^= sub_key[3];

    sub_key[0] = w[2] >> 24;
    sub_key[1] = w[2] >> 16;
    sub_key[2] = w[2] >> 8;
    sub_key[3] = w[2];
```

```
    state[0][2] ^= sub_key[0];
    state[1][2] ^= sub_key[1];
    state[2][2] ^= sub_key[2];
    state[3][2] ^= sub_key[3];

    sub_key[0] = w[3] >> 24;
    sub_key[1] = w[3] >> 16;
    sub_key[2] = w[3] >> 8;
    sub_key[3] = w[3];
    state[0][3] ^= sub_key[0];
    state[1][3] ^= sub_key[1];
    state[2][3] ^= sub_key[2];
    state[3][3] ^= sub_key[3];
}

void SubBytes(uchar state[][4]){
    state[0][0] = sbox[state[0][0] >> 4][state[0][0] & 0x0F];
    state[0][1] = sbox[state[0][1] >> 4][state[0][1] & 0x0F];
    state[0][2] = sbox[state[0][2] >> 4][state[0][2] & 0x0F];
    state[0][3] = sbox[state[0][3] >> 4][state[0][3] & 0x0F];
    state[1][0] = sbox[state[1][0] >> 4][state[1][0] & 0x0F];
    state[1][1] = sbox[state[1][1] >> 4][state[1][1] & 0x0F];
    state[1][2] = sbox[state[1][2] >> 4][state[1][2] & 0x0F];
    state[1][3] = sbox[state[1][3] >> 4][state[1][3] & 0x0F];
    state[2][0] = sbox[state[2][0] >> 4][state[2][0] & 0x0F];
    state[2][1] = sbox[state[2][1] >> 4][state[2][1] & 0x0F];
    state[2][2] = sbox[state[2][2] >> 4][state[2][2] & 0x0F];
    state[2][3] = sbox[state[2][3] >> 4][state[2][3] & 0x0F];
```

```
    state[3][0] = sbox[state[3][0] >> 4][state[3][0] & 0x0F];
    state[3][1] = sbox[state[3][1] >> 4][state[3][1] & 0x0F];
    state[3][2] = sbox[state[3][2] >> 4][state[3][2] & 0x0F];
    state[3][3] = sbox[state[3][3] >> 4][state[3][3] & 0x0F];
}

void ShiftRows(uchar state[][4]){
    int t;

    t = state[1][0];
    state[1][0] = state[1][1];
    state[1][1] = state[1][2];
    state[1][2] = state[1][3];
    state[1][3] = t;

    t = state[2][0];
    state[2][0] = state[2][2];
    state[2][2] = t;
    t = state[2][1];
    state[2][1] = state[2][3];
    state[2][3] = t;

    t = state[3][0];
    state[3][0] = state[3][3];
    state[3][3] = state[3][2];
    state[3][2] = state[3][1];
    state[3][1] = t;
}

void MixColumns(uchar state[][4]){
    uchar column[4];

    column[0] = state[0][0];
    column[1] = state[1][0];
    column[2] = state[2][0];
    column[3] = state[3][0];
```

```
state[0][0] = gf_mul[column[0]][0];
state[0][0] ^= gf_mul[column[1]][1];
state[0][0] ^= column[2];
state[0][0] ^= column[3];
state[1][0] = column[0];
state[1][0] ^= gf_mul[column[1]][0];
state[1][0] ^= gf_mul[column[2]][1];
state[1][0] ^= column[3];
state[2][0] = column[0];
state[2][0] ^= column[1];
state[2][0] ^= gf_mul[column[2]][0];
state[2][0] ^= gf_mul[column[3]][1];
state[3][0] = gf_mul[column[0]][1];
state[3][0] ^= column[1];
state[3][0] ^= column[2];
state[3][0] ^= gf_mul[column[3]][0];

column[0] = state[0][1];
column[1] = state[1][1];
column[2] = state[2][1];
column[3] = state[3][1];
state[0][1] = gf_mul[column[0]][0];
state[0][1] ^= gf_mul[column[1]][1];
state[0][1] ^= column[2];
state[0][1] ^= column[3];
state[1][1] = column[0];
state[1][1] ^= gf_mul[column[1]][0];
state[1][1] ^= gf_mul[column[2]][1];
state[1][1] ^= column[3];
state[2][1] = column[0];
state[2][1] ^= column[1];
state[2][1] ^= gf_mul[column[2]][0];
state[2][1] ^= gf_mul[column[3]][1];
state[3][1] = gf_mul[column[0]][1];
state[3][1] ^= column[1];
state[3][1] ^= column[2];
state[3][1] ^= gf_mul[column[3]][0];
```

```
column[0] = state[0][2];
column[1] = state[1][2];
column[2] = state[2][2];
column[3] = state[3][2];
state[0][2] = gf_mul[column[0]][0];
state[0][2] ^= gf_mul[column[1]][1];
state[0][2] ^= column[2];
state[0][2] ^= column[3];
state[1][2] = column[0];
state[1][2] ^= gf_mul[column[1]][0];
state[1][2] ^= gf_mul[column[2]][1];
state[1][2] ^= column[3];
state[2][2] = column[0];
state[2][2] ^= column[1];
state[2][2] ^= gf_mul[column[2]][0];
state[2][2] ^= gf_mul[column[3]][1];
state[3][2] = gf_mul[column[0]][1];
state[3][2] ^= column[1];
state[3][2] ^= column[2];
state[3][2] ^= gf_mul[column[3]][0];

column[0] = state[0][3];
column[1] = state[1][3];
column[2] = state[2][3];
column[3] = state[3][3];
state[0][3] = gf_mul[column[0]][0];
state[0][3] ^= gf_mul[column[1]][1];
state[0][3] ^= column[2];
state[0][3] ^= column[3];
state[1][3] = column[0];
state[1][3] ^= gf_mul[column[1]][0];
state[1][3] ^= gf_mul[column[2]][1];
state[1][3] ^= column[3];
state[2][3] = column[0];
state[2][3] ^= column[1];
state[2][3] ^= gf_mul[column[2]][0];
state[2][3] ^= gf_mul[column[3]][1];
```

```
    state[3][3] = gf_mul[column[0]][1];
    state[3][3] ^= column[1];
    state[3][3] ^= column[2];
    state[3][3] ^= gf_mul[column[3]][0];
}

uint SubWord(uint word){
    unsigned int result_word;

    result_word = (int)sbox[(word >> 4) & 0x0000000F][word & 0x0000000F];
    result_word += (int)sbox[(word >> 12) & 0x0000000F][(word >> 8) & 0x0000000F] << 8;
    result_word += (int)sbox[(word >> 20) & 0x0000000F][(word >> 16) & 0x0000000F] << 16;
    result_word += (int)sbox[(word >> 28) & 0x0000000F][(word >> 24) & 0x0000000F] << 24;
    return(result_word);
}

void aes_encrypt(uchar input[], uchar output[], uint key[], int keysize){
    uchar temp_state[4][4];
    temp_state[0][0] = input[0];
    temp_state[1][0] = input[1];
    temp_state[2][0] = input[2];
    temp_state[3][0] = input[3];
    temp_state[0][1] = input[4];
    temp_state[1][1] = input[5];
    temp_state[2][1] = input[6];
    temp_state[3][1] = input[7];
    temp_state[0][2] = input[8];
    temp_state[1][2] = input[9];
    temp_state[2][2] = input[10];
    temp_state[3][2] = input[11];
    temp_state[0][3] = input[12];
    temp_state[1][3] = input[13];
    temp_state[2][3] = input[14];
    temp_state[3][3] = input[15];

    AddRoundKey(temp_state,&key[0]);
    SubBytes(temp_state); ShiftRows(temp_state); MixColumns(temp_state); AddRoundKey(temp_state,&key[4]);
    SubBytes(temp_state); ShiftRows(temp_state); MixColumns(temp_state); AddRoundKey(temp_state,&key[8]);
```

```
    SubBytes(temp_state); ShiftRows(temp_state); MixColumns(temp_state); AddRoundKey(temp_state,&key[12]);
    SubBytes(temp_state); ShiftRows(temp_state); MixColumns(temp_state); AddRoundKey(temp_state,&key[16]);
    SubBytes(temp_state); ShiftRows(temp_state); MixColumns(temp_state); AddRoundKey(temp_state,&key[20]);
    SubBytes(temp_state); ShiftRows(temp_state); MixColumns(temp_state); AddRoundKey(temp_state,&key[24]);
    SubBytes(temp_state); ShiftRows(temp_state); MixColumns(temp_state); AddRoundKey(temp_state,&key[28]);
    SubBytes(temp_state); ShiftRows(temp_state); MixColumns(temp_state); AddRoundKey(temp_state,&key[32]);
    SubBytes(temp_state); ShiftRows(temp_state); MixColumns(temp_state); AddRoundKey(temp_state,&key[36]);
    if (keysize != 128) {
        SubBytes(temp_state); ShiftRows(temp_state); MixColumns(temp_state); AddRoundKey(temp_state,&key[40]);
        SubBytes(temp_state); ShiftRows(temp_state); MixColumns(temp_state); AddRoundKey(temp_state,&key[44]);
        if (keysize != 192) {
            SubBytes(temp_state); ShiftRows(temp_state); MixColumns(temp_state); AddRoundKey(temp_state,&key[48]);
            SubBytes(temp_state); ShiftRows(temp_state); MixColumns(temp_state); AddRoundKey(temp_state,&key[52]);
            SubBytes(temp_state); ShiftRows(temp_state); AddRoundKey(temp_state,&key[56]);
        }
        else {
            SubBytes(temp_state); ShiftRows(temp_state); AddRoundKey(temp_state,&key[48]);
        }
    }
    else {
        SubBytes(temp_state); ShiftRows(temp_state); AddRoundKey(temp_state,&key[40]);
    }
    output[0] = temp_state[0][0];
    output[1] = temp_state[1][0];
    output[2] = temp_state[2][0];
    output[3] = temp_state[3][0];
    output[4] = temp_state[0][1];
    output[5] = temp_state[1][1];
    output[6] = temp_state[2][1];
    output[7] = temp_state[3][1];
    output[8] = temp_state[0][2];
    output[9] = temp_state[1][2];
    output[10] = temp_state[2][2];
    output[11] = temp_state[3][2];
    output[12] = temp_state[0][3];
    output[13] = temp_state[1][3];
    output[14] = temp_state[2][3];
    output[15] = temp_state[3][3];
}
```

```
void XTS_AES_block(uchar tweak[], uchar plaintext[], uint key1[],uint key2[], int i){
    aes_encrypt(tweak,ciphertext_tweak,key_schedule2,256);

    jj=j;

    while(jj!=0){
        LCheckBit=0;
        for (idxx=0; idxx < 16; idxx++) {
            MCheckBit = (ciphertext_tweak[idxx] >> 7);
            ciphertext_tweak[idxx] = (ciphertext_tweak[idxx] << 1)& 0xff;

            if(LCheckBit == 1){
                ciphertext_tweak[idxx]=ciphertext_tweak[idxx]^0x01;
            }

            LCheckBit = MCheckBit;
            if (MCheckBit == 1 && idxx == 15){
                ciphertext_tweak[0]=ciphertext_tweak[0]^0x87;
            }
        }
        jj--;
    }

    for (idx=0; idx < 16; idx++){
        T[idx]=ciphertext_tweak[idx];
        PP[idx]= T[idx]^plaintext[idx];
    }

    aes_encrypt(PP,CC,key_schedule1,256);

    for (idx=0; idx < 16; idx++){
        ciphertexts[i][idx]= T[idx]^CC[idx];
    }
}

#define KE_ROTWORD(z) ( ((z) << 8) | ((z) >> 24) )
```

```
void KeyExpansion(uchar key[], uint w[], int keysize){
   int Nb=4,Nr,Nk,i;
   uint temp,Rcon[]={0x01000000,0x02000000,0x04000000,0x08000000,0x10000000,0x20000000,
                     0x40000000,0x80000000,0x1b000000,0x36000000,0x6c000000,0xd8000000,
                     0xab000000,0x4d000000,0x9a000000};
   switch (keysize) {
      case 128: Nr = 10; Nk = 4; break;
      case 192: Nr = 12; Nk = 6; break;
      case 256: Nr = 14; Nk = 8; break;
      default: return;
   }

   for (i=0; i < Nk; ++i) {
      w[i] = ((key[4 * i]) << 24) | ((key[4 * i + 1]) << 16) |
             ((key[4 * i + 2]) << 8) | ((key[4 * i + 3]));
   }

   for (i = Nk; i < Nb * (Nr+1); ++i) {
      temp = w[i - 1];
      if ((i % Nk) == 0)
         temp = SubWord(KE_ROTWORD(temp)) ^ Rcon[(i-1)/Nk];
      else if (Nk > 6 && (i % Nk) == 4)
         temp = SubWord(temp);
      w[i] = w[i-Nk] ^ temp;
   }
}


/*~~~~~~~~~~~~~~~~~~~~~~~~~~~MAIN FUNCTION~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
int main(int argc, char **argv) {
    clock_t begin, end;
    double time_spent;
    begin = clock();
    int nprocs, myrank;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

```
    int i, j, k, s=15, l=0;
    static uchar global_input[TOTAL_NUM_OF_ITEMS/2];uchar local_input[numi];
    static uchar global_tweak[TOTAL_NUM_OF_TWEAKS/2], local_tweak[numi];
    uchar line[3];

    uchar key1[32] = {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,
                0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f,0x10,0x11,
                0x12,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1a,
                0x1b,0x1c,0x1d,0x1e,0x1f},
          key2[32] = {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,
                0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f,0x10,0x11,
                0x12,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1a,
                0x1b,0x1c,0x1d,0x1e,0x1f};

    FILE * fi, * ft, * fo;

/*~~~~~~~~~~~~~~~~~PROCESSOR ZERO INITIAL ACTIONS~~~~~~~~~~~~~~~~*/
    if (myrank == 0) {
        fi = fopen("input.txt", "rt");
        if (fi == NULL){
            printf("could not open the input file. \n\n");
            exit(1);
        }
        for (i=0; i<TOTAL_NUM_OF_ITEMS/2; i++){
                if(fgets(line, 3, fi) != NULL){
                sscanf(line, "%02x", &global_input[i]);
            }
            else {
                global_input[i]=0;
            }
        }

        fo = fopen("p1.txt","wt");
        for(i=0;i<TOTAL_NUM_OF_ITEMS/2;i++){
            fprintf(fo,"%02x",global_input[i]);
        }

        ft = fopen("tweak.txt", "rt");
        if (ft == NULL){
```

```
            printf("could not open the input tweak file. \n\n");
            exit(1);
        }
        for (i=0; i<TOTAL_NUM_OF_TWEAKS/2; i++){
            if(fgets(line, 3, ft) != NULL){
            sscanf(line, "%02x", &global_tweak[i]);
            }
            else {
                global_tweak[i]=0;
            }
        }

        fclose(fo);
        fclose(fi);
        fclose(ft);

        //expanding key1 and key2
        KeyExpansion(key1,key_schedule1,256);
        KeyExpansion(key2,key_schedule2,256);
    }

    MPI_Scatter(global_input, numi, MPI_UNSIGNED_CHAR,&local_input, numi,
            MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);

    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Scatter(global_tweak, numi, MPI_UNSIGNED_CHAR, &local_tweak, numi,
            MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);

    MPI_Barrier(MPI_COMM_WORLD);

    //putting input plain-texts into 2D array
    k = 0;
    for (i=0; i<numb; i++) {
        for(j=0;j<16;j++){
            plaintexts[i][j] = local_input[k];
            k++;
        }
    }
```

```
//putting tweaks into 2D array
k = 0;
for (i=0; i<numb; i++) {
    for(j=0;j<16;j++){
        tweaks[i][j] = local_tweak[k];
        k++;
    }
}

//processing each block
j = numb * myrank;
for (i =0; i <numb; i++) {
    j = j+1;
    for(k=0;k<16;k++){
        if (i != numb)  {
            plaintext[k] = plaintexts[i][k];
        } else{
            if(plaintexts[i][k]!='\0'){
            s=k;
            plaintext[k]=plaintexts[i][k];
          } else{
            plaintext[k]=ciphertexts[i-1][k];
            }
        }
        tweak[k]=tweaks[i][k];
    }

if(s == 15){
    XTS_AES_block(tweak, plaintext, key_schedule1, key_schedule2,i);
}else {
    while(l<s || l ==s){
        ciphertexts[i][l]=ciphertexts[i-1][l];
        l++;
    }
    for(l=s+1;l<16;l++){
        ciphertexts[i][l]='\0';
    }
    XTS_AES_block(tweak, plaintext,key_schedule1, key_schedule2,i-1);
}
```

```
    }

    k=0;
    for(i=0; i<numb; i++){
        for(j=0; j<16; j++){
            local_input[k] = ciphertexts[i][j];
            k++;
        }
    }

    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Gather(&local_input, numi, MPI_UNSIGNED_CHAR, global_input, numi,
        MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);

    if (myrank == 0) {
            fo = fopen("output.txt","wt");

        for(i=0; i<TOTAL_NUM_OF_ITEMS/2; i++){
            fprintf(fo,"%02x",global_input[i]);
        }
        fclose(fo);

    end = clock();
    time_spent=(double)(end-begin)/CLOCKS_PER_SEC;
    printf("Total time = %f\n\n",time_spent);
    }

    MPI_Finalize();
        return 0;
}
```

[B] XTS_AES_Decrypt.c

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

#define TOTAL_NUM_OF_ITEMS 16777216
#define TOTAL_NUM_OF_TWEAKS 16777216
#define numb 32768
#define numi 524288

#define uchar unsigned char // 8 bits
#define uint unsigned long // 32 bits


int idx, LCheckBit, MCheckBit, j=1, jj, k, idx, idxx, first=1;
uchar tweaks[numb][16], tweak[16];
uchar plaintexts[numb][16], plaintext[16], second_last_ciphertext[16];
uchar ciphertexts[numb][16], ciphertext[16], ciphertext_tweak[16] ;
uchar T[16], PP[16], CC[16], temp[16];
uint key_schedule1[60],key_schedule2[60];

uchar sbox[16][16] = {
    0x63,0x7C,0x77,0x7B,0xF2,0x6B,0x6F,0xC5,0x30,0x01,0x67,0x2B,0xFE,0xD7,0xAB,0x76,
    0xCA,0x82,0xC9,0x7D,0xFA,0x59,0x47,0xF0,0xAD,0xD4,0xA2,0xAF,0x9C,0xA4,0x72,0xC0,
    0xB7,0xFD,0x93,0x26,0x36,0x3F,0xF7,0xCC,0x34,0xA5,0xE5,0xF1,0x71,0xD8,0x31,0x15,
    0x04,0xC7,0x23,0xC3,0x18,0x96,0x05,0x9A,0x07,0x12,0x80,0xE2,0xEB,0x27,0xB2,0x75,
    0x09,0x83,0x2C,0x1A,0x1B,0x6E,0x5A,0xA0,0x52,0x3B,0xD6,0xB3,0x29,0xE3,0x2F,0x84,
    0x53,0xD1,0x00,0xED,0x20,0xFC,0xB1,0x5B,0x6A,0xCB,0xBE,0x39,0x4A,0x4C,0x58,0xCF,
    0xD0,0xEF,0xAA,0xFB,0x43,0x4D,0x33,0x85,0x45,0xF9,0x02,0x7F,0x50,0x3C,0x9F,0xA8,
    0x51,0xA3,0x40,0x8F,0x92,0x9D,0x38,0xF5,0xBC,0xB6,0xDA,0x21,0x10,0xFF,0xF3,0xD2,
    0xCD,0x0C,0x13,0xEC,0x5F,0x97,0x44,0x17,0xC4,0xA7,0x7E,0x3D,0x64,0x5D,0x19,0x73,
    0x60,0x81,0x4F,0xDC,0x22,0x2A,0x90,0x88,0x46,0xEE,0xB8,0x14,0xDE,0x5E,0x0B,0xDB,
    0xE0,0x32,0x3A,0x0A,0x49,0x06,0x24,0x5C,0xC2,0xD3,0xAC,0x62,0x91,0x95,0xE4,0x79,
    0xE7,0xC8,0x37,0x6D,0x8D,0xD5,0x4E,0xA9,0x6C,0x56,0xF4,0xEA,0x65,0x7A,0xAE,0x08,
    0xBA,0x78,0x25,0x2E,0x1C,0xA6,0xB4,0xC6,0xE8,0xDD,0x74,0x1F,0x4B,0xBD,0x8B,0x8A,
    0x70,0x3E,0xB5,0x66,0x48,0x03,0xF6,0x0E,0x61,0x35,0x57,0xB9,0x86,0xC1,0x1D,0x9E,
    0xE1,0xF8,0x98,0x11,0x69,0xD9,0x8E,0x94,0x9B,0x1E,0x87,0xE9,0xCE,0x55,0x28,0xDF,
    0x8C,0xA1,0x89,0x0D,0xBF,0xE6,0x42,0x68,0x41,0x99,0x2D,0x0F,0xB0,0x54,0xBB,0x16
};
```

```
uchar inv_sbox[16][16] = {
   0x52,0x09,0x6A,0xD5,0x30,0x36,0xA5,0x38,0xBF,0x40,0xA3,0x9E,0x81,0xF3,0xD7,0xFB,
   0x7C,0xE3,0x39,0x82,0x9B,0x2F,0xFF,0x87,0x34,0x8E,0x43,0x44,0xC4,0xDE,0xE9,0xCB,
   0x54,0x7B,0x94,0x32,0xA6,0xC2,0x23,0x3D,0xEE,0x4C,0x95,0x0B,0x42,0xFA,0xC3,0x4E,
   0x08,0x2E,0xA1,0x66,0x28,0xD9,0x24,0xB2,0x76,0x5B,0xA2,0x49,0x6D,0x8B,0xD1,0x25,
   0x72,0xF8,0xF6,0x64,0x86,0x68,0x98,0x16,0xD4,0xA4,0x5C,0xCC,0x5D,0x65,0xB6,0x92,
   0x6C,0x70,0x48,0x50,0xFD,0xED,0xB9,0xDA,0x5E,0x15,0x46,0x57,0xA7,0x8D,0x9D,0x84,
   0x90,0xD8,0xAB,0x00,0x8C,0xBC,0xD3,0x0A,0xF7,0xE4,0x58,0x05,0xB8,0xB3,0x45,0x06,
   0xD0,0x2C,0x1E,0x8F,0xCA,0x3F,0x0F,0x02,0xC1,0xAF,0xBD,0x03,0x01,0x13,0x8A,0x6B,
   0x3A,0x91,0x11,0x41,0x4F,0x67,0xDC,0xEA,0x97,0xF2,0xCF,0xCE,0xF0,0xB4,0xE6,0x73,
   0x96,0xAC,0x74,0x22,0xE7,0xAD,0x35,0x85,0xE2,0xF9,0x37,0xE8,0x1C,0x75,0xDF,0x6E,
   0x47,0xF1,0x1A,0x71,0x1D,0x29,0xC5,0x89,0x6F,0xB7,0x62,0x0E,0xAA,0x18,0xBE,0x1B,
   0xFC,0x56,0x3E,0x4B,0xC6,0xD2,0x79,0x20,0x9A,0xDB,0xC0,0xFE,0x78,0xCD,0x5A,0xF4,
   0x1F,0xDD,0xA8,0x33,0x88,0x07,0xC7,0x31,0xB1,0x12,0x10,0x59,0x27,0x80,0xEC,0x5F,
   0x60,0x51,0x7F,0xA9,0x19,0xB5,0x4A,0x0D,0x2D,0xE5,0x7A,0x9F,0x93,0xC9,0x9C,0xEF,
   0xA0,0xE0,0x3B,0x4D,0xAE,0x2A,0xF5,0xB0,0xC8,0xEB,0xBB,0x3C,0x83,0x53,0x99,0x61,
   0x17,0x2B,0x04,0x7E,0xBA,0x77,0xD6,0x26,0xE1,0x69,0x14,0x63,0x55,0x21,0x0C,0x7D
};

uchar gf_mul[256][6] = {
   {0x00,0x00,0x00,0x00,0x00,0x00},{0x02,0x03,0x09,0x0b,0x0d,0x0e},
   {0x04,0x06,0x12,0x16,0x1a,0x1c},{0x06,0x05,0x1b,0x1d,0x17,0x12},
   {0x08,0x0c,0x24,0x2c,0x34,0x38},{0x0a,0x0f,0x2d,0x27,0x39,0x36},
   {0x0c,0x0a,0x36,0x3a,0x2e,0x24},{0x0e,0x09,0x3f,0x31,0x23,0x2a},
   {0x10,0x18,0x48,0x58,0x68,0x70},{0x12,0x1b,0x41,0x53,0x65,0x7e},
   {0x14,0x1e,0x5a,0x4e,0x72,0x6c},{0x16,0x1d,0x53,0x45,0x7f,0x62},
   {0x18,0x14,0x6c,0x74,0x5c,0x48},{0x1a,0x17,0x65,0x7f,0x51,0x46},
   {0x1c,0x12,0x7e,0x62,0x46,0x54},{0x1e,0x11,0x77,0x69,0x4b,0x5a},
   {0x20,0x30,0x90,0xb0,0xd0,0xe0},{0x22,0x33,0x99,0xbb,0xdd,0xee},
   {0x24,0x36,0x82,0xa6,0xca,0xfc},{0x26,0x35,0x8b,0xad,0xc7,0xf2},
   {0x28,0x3c,0xb4,0x9c,0xe4,0xd8},{0x2a,0x3f,0xbd,0x97,0xe9,0xd6},
   {0x2c,0x3a,0xa6,0x8a,0xfe,0xc4},{0x2e,0x39,0xaf,0x81,0xf3,0xca},
   {0x30,0x28,0xd8,0xe8,0xb8,0x90},{0x32,0x2b,0xd1,0xe3,0xb5,0x9e},
   {0x34,0x2e,0xca,0xfe,0xa2,0x8c},{0x36,0x2d,0xc3,0xf5,0xaf,0x82},
   {0x38,0x24,0xfc,0xc4,0x8c,0xa8},{0x3a,0x27,0xf5,0xcf,0x81,0xa6},
   {0x3c,0x22,0xee,0xd2,0x96,0xb4},{0x3e,0x21,0xe7,0xd9,0x9b,0xba},
   {0x40,0x60,0x3b,0x7b,0xbb,0xdb},{0x42,0x63,0x32,0x70,0xb6,0xd5},
   {0x44,0x66,0x29,0x6d,0xa1,0xc7},{0x46,0x65,0x20,0x66,0xac,0xc9},
   {0x48,0x6c,0x1f,0x57,0x8f,0xe3},{0x4a,0x6f,0x16,0x5c,0x82,0xed},
```

```
{0x4c,0x6a,0x0d,0x41,0x95,0xff},{0x4e,0x69,0x04,0x4a,0x98,0xf1},
{0x50,0x78,0x73,0x23,0xd3,0xab},{0x52,0x7b,0x7a,0x28,0xde,0xa5},
{0x54,0x7e,0x61,0x35,0xc9,0xb7},{0x56,0x7d,0x68,0x3e,0xc4,0xb9},
{0x58,0x74,0x57,0x0f,0xe7,0x93},{0x5a,0x77,0x5e,0x04,0xea,0x9d},
{0x5c,0x72,0x45,0x19,0xfd,0x8f},{0x5e,0x71,0x4c,0x12,0xf0,0x81},
{0x60,0x50,0xab,0xcb,0x6b,0x3b},{0x62,0x53,0xa2,0xc0,0x66,0x35},
{0x64,0x56,0xb9,0xdd,0x71,0x27},{0x66,0x55,0xb0,0xd6,0x7c,0x29},
{0x68,0x5c,0x8f,0xe7,0x5f,0x03},{0x6a,0x5f,0x86,0xec,0x52,0x0d},
{0x6c,0x5a,0x9d,0xf1,0x45,0x1f},{0x6e,0x59,0x94,0xfa,0x48,0x11},
{0x70,0x48,0xe3,0x93,0x03,0x4b},{0x72,0x4b,0xea,0x98,0x0e,0x45},
{0x74,0x4e,0xf1,0x85,0x19,0x57},{0x76,0x4d,0xf8,0x8e,0x14,0x59},
{0x78,0x44,0xc7,0xbf,0x37,0x73},{0x7a,0x47,0xce,0xb4,0x3a,0x7d},
{0x7c,0x42,0xd5,0xa9,0x2d,0x6f},{0x7e,0x41,0xdc,0xa2,0x20,0x61},
{0x80,0xc0,0x76,0xf6,0x6d,0xad},{0x82,0xc3,0x7f,0xfd,0x60,0xa3},
{0x84,0xc6,0x64,0xe0,0x77,0xb1},{0x86,0xc5,0x6d,0xeb,0x7a,0xbf},
{0x88,0xcc,0x52,0xda,0x59,0x95},{0x8a,0xcf,0x5b,0xd1,0x54,0x9b},
{0x8c,0xca,0x40,0xcc,0x43,0x89},{0x8e,0xc9,0x49,0xc7,0x4e,0x87},
{0x90,0xd8,0x3e,0xae,0x05,0xdd},{0x92,0xdb,0x37,0xa5,0x08,0xd3},
{0x94,0xde,0x2c,0xb8,0x1f,0xc1},{0x96,0xdd,0x25,0xb3,0x12,0xcf},
{0x98,0xd4,0x1a,0x82,0x31,0xe5},{0x9a,0xd7,0x13,0x89,0x3c,0xeb},
{0x9c,0xd2,0x08,0x94,0x2b,0xf9},{0x9e,0xd1,0x01,0x9f,0x26,0xf7},
{0xa0,0xf0,0xe6,0x46,0xbd,0x4d},{0xa2,0xf3,0xef,0x4d,0xb0,0x43},
{0xa4,0xf6,0xf4,0x50,0xa7,0x51},{0xa6,0xf5,0xfd,0x5b,0xaa,0x5f},
{0xa8,0xfc,0xc2,0x6a,0x89,0x75},{0xaa,0xff,0xcb,0x61,0x84,0x7b},
{0xac,0xfa,0xd0,0x7c,0x93,0x69},{0xae,0xf9,0xd9,0x77,0x9e,0x67},
{0xb0,0xe8,0xae,0x1e,0xd5,0x3d},{0xb2,0xeb,0xa7,0x15,0xd8,0x33},
{0xb4,0xee,0xbc,0x08,0xcf,0x21},{0xb6,0xed,0xb5,0x03,0xc2,0x2f},
{0xb8,0xe4,0x8a,0x32,0xe1,0x05},{0xba,0xe7,0x83,0x39,0xec,0x0b},
{0xbc,0xe2,0x98,0x24,0xfb,0x19},{0xbe,0xe1,0x91,0x2f,0xf6,0x17},
{0xc0,0xa0,0x4d,0x8d,0xd6,0x76},{0xc2,0xa3,0x44,0x86,0xdb,0x78},
{0xc4,0xa6,0x5f,0x9b,0xcc,0x6a},{0xc6,0xa5,0x56,0x90,0xc1,0x64},
{0xc8,0xac,0x69,0xa1,0xe2,0x4e},{0xca,0xaf,0x60,0xaa,0xef,0x40},
{0xcc,0xaa,0x7b,0xb7,0xf8,0x52},{0xce,0xa9,0x72,0xbc,0xf5,0x5c},
{0xd0,0xb8,0x05,0xd5,0xbe,0x06},{0xd2,0xbb,0x0c,0xde,0xb3,0x08},
{0xd4,0xbe,0x17,0xc3,0xa4,0x1a},{0xd6,0xbd,0x1e,0xc8,0xa9,0x14},
{0xd8,0xb4,0x21,0xf9,0x8a,0x3e},{0xda,0xb7,0x28,0xf2,0x87,0x30},
{0xdc,0xb2,0x33,0xef,0x90,0x22},{0xde,0xb1,0x3a,0xe4,0x9d,0x2c},
{0xe0,0x90,0xdd,0x3d,0x06,0x96},{0xe2,0x93,0xd4,0x36,0x0b,0x98},
{0xe4,0x96,0xcf,0x2b,0x1c,0x8a},{0xe6,0x95,0xc6,0x20,0x11,0x84},
```

```
{0xe8,0x9c,0xf9,0x11,0x32,0xae},{0xea,0x9f,0xf0,0x1a,0x3f,0xa0},
{0xec,0x9a,0xeb,0x07,0x28,0xb2},{0xee,0x99,0xe2,0x0c,0x25,0xbc},
{0xf0,0x88,0x95,0x65,0x6e,0xe6},{0xf2,0x8b,0x9c,0x6e,0x63,0xe8},
{0xf4,0x8e,0x87,0x73,0x74,0xfa},{0xf6,0x8d,0x8e,0x78,0x79,0xf4},
{0xf8,0x84,0xb1,0x49,0x5a,0xde},{0xfa,0x87,0xb8,0x42,0x57,0xd0},
{0xfc,0x82,0xa3,0x5f,0x40,0xc2},{0xfe,0x81,0xaa,0x54,0x4d,0xcc},
{0x1b,0x9b,0xec,0xf7,0xda,0x41},{0x19,0x98,0xe5,0xfc,0xd7,0x4f},
{0x1f,0x9d,0xfe,0xe1,0xc0,0x5d},{0x1d,0x9e,0xf7,0xea,0xcd,0x53},
{0x13,0x97,0xc8,0xdb,0xee,0x79},{0x11,0x94,0xc1,0xd0,0xe3,0x77},
{0x17,0x91,0xda,0xcd,0xf4,0x65},{0x15,0x92,0xd3,0xc6,0xf9,0x6b},
{0x0b,0x83,0xa4,0xaf,0xb2,0x31},{0x09,0x80,0xad,0xa4,0xbf,0x3f},
{0x0f,0x85,0xb6,0xb9,0xa8,0x2d},{0x0d,0x86,0xbf,0xb2,0xa5,0x23},
{0x03,0x8f,0x80,0x83,0x86,0x09},{0x01,0x8c,0x89,0x88,0x8b,0x07},
{0x07,0x89,0x92,0x95,0x9c,0x15},{0x05,0x8a,0x9b,0x9e,0x91,0x1b},
{0x3b,0xab,0x7c,0x47,0x0a,0xa1},{0x39,0xa8,0x75,0x4c,0x07,0xaf},
{0x3f,0xad,0x6e,0x51,0x10,0xbd},{0x3d,0xae,0x67,0x5a,0x1d,0xb3},
{0x33,0xa7,0x58,0x6b,0x3e,0x99},{0x31,0xa4,0x51,0x60,0x33,0x97},
{0x37,0xa1,0x4a,0x7d,0x24,0x85},{0x35,0xa2,0x43,0x76,0x29,0x8b},
{0x2b,0xb3,0x34,0x1f,0x62,0xd1},{0x29,0xb0,0x3d,0x14,0x6f,0xdf},
{0x2f,0xb5,0x26,0x09,0x78,0xcd},{0x2d,0xb6,0x2f,0x02,0x75,0xc3},
{0x23,0xbf,0x10,0x33,0x56,0xe9},{0x21,0xbc,0x19,0x38,0x5b,0xe7},
{0x27,0xb9,0x02,0x25,0x4c,0xf5},{0x25,0xba,0x0b,0x2e,0x41,0xfb},
{0x5b,0xfb,0xd7,0x8c,0x61,0x9a},{0x59,0xf8,0xde,0x87,0x6c,0x94},
{0x5f,0xfd,0xc5,0x9a,0x7b,0x86},{0x5d,0xfe,0xcc,0x91,0x76,0x88},
{0x53,0xf7,0xf3,0xa0,0x55,0xa2},{0x51,0xf4,0xfa,0xab,0x58,0xac},
{0x57,0xf1,0xe1,0xb6,0x4f,0xbe},{0x55,0xf2,0xe8,0xbd,0x42,0xb0},
{0x4b,0xe3,0x9f,0xd4,0x09,0xea},{0x49,0xe0,0x96,0xdf,0x04,0xe4},
{0x4f,0xe5,0x8d,0xc2,0x13,0xf6},{0x4d,0xe6,0x84,0xc9,0x1e,0xf8},
{0x43,0xef,0xbb,0xf8,0x3d,0xd2},{0x41,0xec,0xb2,0xf3,0x30,0xdc},
{0x47,0xe9,0xa9,0xee,0x27,0xce},{0x45,0xea,0xa0,0xe5,0x2a,0xc0},
{0x7b,0xcb,0x47,0x3c,0xb1,0x7a},{0x79,0xc8,0x4e,0x37,0xbc,0x74},
{0x7f,0xcd,0x55,0x2a,0xab,0x66},{0x7d,0xce,0x5c,0x21,0xa6,0x68},
{0x73,0xc7,0x63,0x10,0x85,0x42},{0x71,0xc4,0x6a,0x1b,0x88,0x4c},
{0x77,0xc1,0x71,0x06,0x9f,0x5e},{0x75,0xc2,0x78,0x0d,0x92,0x50},
{0x6b,0xd3,0x0f,0x64,0xd9,0x0a},{0x69,0xd0,0x06,0x6f,0xd4,0x04},
{0x6f,0xd5,0x1d,0x72,0xc3,0x16},{0x6d,0xd6,0x14,0x79,0xce,0x18},
{0x63,0xdf,0x2b,0x48,0xed,0x32},{0x61,0xdc,0x22,0x43,0xe0,0x3c},
{0x67,0xd9,0x39,0x5e,0xf7,0x2e},{0x65,0xda,0x30,0x55,0xfa,0x20},
{0x9b,0x5b,0x9a,0x01,0xb7,0xec},{0x99,0x58,0x93,0x0a,0xba,0xe2},
```

```
    {0x9f,0x5d,0x88,0x17,0xad,0xf0},{0x9d,0x5e,0x81,0x1c,0xa0,0xfe},
    {0x93,0x57,0xbe,0x2d,0x83,0xd4},{0x91,0x54,0xb7,0x26,0x8e,0xda},
    {0x97,0x51,0xac,0x3b,0x99,0xc8},{0x95,0x52,0xa5,0x30,0x94,0xc6},
    {0x8b,0x43,0xd2,0x59,0xdf,0x9c},{0x89,0x40,0xdb,0x52,0xd2,0x92},
    {0x8f,0x45,0xc0,0x4f,0xc5,0x80},{0x8d,0x46,0xc9,0x44,0xc8,0x8e},
    {0x83,0x4f,0xf6,0x75,0xeb,0xa4},{0x81,0x4c,0xff,0x7e,0xe6,0xaa},
    {0x87,0x49,0xe4,0x63,0xf1,0xb8},{0x85,0x4a,0xed,0x68,0xfc,0xb6},
    {0xbb,0x6b,0x0a,0xb1,0x67,0x0c},{0xb9,0x68,0x03,0xba,0x6a,0x02},
    {0xbf,0x6d,0x18,0xa7,0x7d,0x10},{0xbd,0x6e,0x11,0xac,0x70,0x1e},
    {0xb3,0x67,0x2e,0x9d,0x53,0x34},{0xb1,0x64,0x27,0x96,0x5e,0x3a},
    {0xb7,0x61,0x3c,0x8b,0x49,0x28},{0xb5,0x62,0x35,0x80,0x44,0x26},
    {0xab,0x73,0x42,0xe9,0x0f,0x7c},{0xa9,0x70,0x4b,0xe2,0x02,0x72},
    {0xaf,0x75,0x50,0xff,0x15,0x60},{0xad,0x76,0x59,0xf4,0x18,0x6e},
    {0xa3,0x7f,0x66,0xc5,0x3b,0x44},{0xa1,0x7c,0x6f,0xce,0x36,0x4a},
    {0xa7,0x79,0x74,0xd3,0x21,0x58},{0xa5,0x7a,0x7d,0xd8,0x2c,0x56},
    {0xdb,0x3b,0xa1,0x7a,0x0c,0x37},{0xd9,0x38,0xa8,0x71,0x01,0x39},
    {0xdf,0x3d,0xb3,0x6c,0x16,0x2b},{0xdd,0x3e,0xba,0x67,0x1b,0x25},
    {0xd3,0x37,0x85,0x56,0x38,0x0f},{0xd1,0x34,0x8c,0x5d,0x35,0x01},
    {0xd7,0x31,0x97,0x40,0x22,0x13},{0xd5,0x32,0x9e,0x4b,0x2f,0x1d},
    {0xcb,0x23,0xe9,0x22,0x64,0x47},{0xc9,0x20,0xe0,0x29,0x69,0x49},
    {0xcf,0x25,0xfb,0x34,0x7e,0x5b},{0xcd,0x26,0xf2,0x3f,0x73,0x55},
    {0xc3,0x2f,0xcd,0x0e,0x50,0x7f},{0xc1,0x2c,0xc4,0x05,0x5d,0x71},
    {0xc7,0x29,0xdf,0x18,0x4a,0x63},{0xc5,0x2a,0xd6,0x13,0x47,0x6d},
    {0xfb,0x0b,0x31,0xca,0xdc,0xd7},{0xf9,0x08,0x38,0xc1,0xd1,0xd9},
    {0xff,0x0d,0x23,0xdc,0xc6,0xcb},{0xfd,0x0e,0x2a,0xd7,0xcb,0xc5},
    {0xf3,0x07,0x15,0xe6,0xe8,0xef},{0xf1,0x04,0x1c,0xed,0xe5,0xe1},
    {0xf7,0x01,0x07,0xf0,0xf2,0xf3},{0xf5,0x02,0x0e,0xfb,0xff,0xfd},
    {0xeb,0x13,0x79,0x92,0xb4,0xa7},{0xe9,0x10,0x70,0x99,0xb9,0xa9},
    {0xef,0x15,0x6b,0x84,0xae,0xbb},{0xed,0x16,0x62,0x8f,0xa3,0xb5},
    {0xe3,0x1f,0x5d,0xbe,0x80,0x9f},{0xe1,0x1c,0x54,0xb5,0x8d,0x91},
    {0xe7,0x19,0x4f,0xa8,0x9a,0x83},{0xe5,0x1a,0x46,0xa3,0x97,0x8d}
};

void AddRoundKey(uchar state[][4], uint w[]){
    uchar sub_key[4];

    sub_key[0] = w[0] >> 24;
    sub_key[1] = w[0] >> 16;
    sub_key[2] = w[0] >> 8;
```

```
    sub_key[3] = w[0];
    state[0][0] ^= sub_key[0];
    state[1][0] ^= sub_key[1];
    state[2][0] ^= sub_key[2];
    state[3][0] ^= sub_key[3];

    sub_key[0] = w[1] >> 24;
    sub_key[1] = w[1] >> 16;
    sub_key[2] = w[1] >> 8;
    sub_key[3] = w[1];
    state[0][1] ^= sub_key[0];
    state[1][1] ^= sub_key[1];
    state[2][1] ^= sub_key[2];
    state[3][1] ^= sub_key[3];

    sub_key[0] = w[2] >> 24;
    sub_key[1] = w[2] >> 16;
    sub_key[2] = w[2] >> 8;
    sub_key[3] = w[2];
    state[0][2] ^= sub_key[0];
    state[1][2] ^= sub_key[1];
    state[2][2] ^= sub_key[2];
    state[3][2] ^= sub_key[3];

    sub_key[0] = w[3] >> 24;
    sub_key[1] = w[3] >> 16;
    sub_key[2] = w[3] >> 8;
    sub_key[3] = w[3];
    state[0][3] ^= sub_key[0];
    state[1][3] ^= sub_key[1];
    state[2][3] ^= sub_key[2];
    state[3][3] ^= sub_key[3];
}

void SubBytes(uchar state[][4]){
    state[0][0] = sbox[state[0][0] >> 4][state[0][0] & 0x0F];
    state[0][1] = sbox[state[0][1] >> 4][state[0][1] & 0x0F];
    state[0][2] = sbox[state[0][2] >> 4][state[0][2] & 0x0F];
    state[0][3] = sbox[state[0][3] >> 4][state[0][3] & 0x0F];
```

```
    state[1][0] = sbox[state[1][0] >> 4][state[1][0] & 0x0F];
    state[1][1] = sbox[state[1][1] >> 4][state[1][1] & 0x0F];
    state[1][2] = sbox[state[1][2] >> 4][state[1][2] & 0x0F];
    state[1][3] = sbox[state[1][3] >> 4][state[1][3] & 0x0F];
    state[2][0] = sbox[state[2][0] >> 4][state[2][0] & 0x0F];
    state[2][1] = sbox[state[2][1] >> 4][state[2][1] & 0x0F];
    state[2][2] = sbox[state[2][2] >> 4][state[2][2] & 0x0F];
    state[2][3] = sbox[state[2][3] >> 4][state[2][3] & 0x0F];
    state[3][0] = sbox[state[3][0] >> 4][state[3][0] & 0x0F];
    state[3][1] = sbox[state[3][1] >> 4][state[3][1] & 0x0F];
    state[3][2] = sbox[state[3][2] >> 4][state[3][2] & 0x0F];
    state[3][3] = sbox[state[3][3] >> 4][state[3][3] & 0x0F];
}

void Inv_SubBytes(uchar state[][4]){
    state[0][0] = inv_sbox[state[0][0] >> 4][state[0][0] & 0x0F];
    state[0][1] = inv_sbox[state[0][1] >> 4][state[0][1] & 0x0F];
    state[0][2] = inv_sbox[state[0][2] >> 4][state[0][2] & 0x0F];
    state[0][3] = inv_sbox[state[0][3] >> 4][state[0][3] & 0x0F];
    state[1][0] = inv_sbox[state[1][0] >> 4][state[1][0] & 0x0F];
    state[1][1] = inv_sbox[state[1][1] >> 4][state[1][1] & 0x0F];
    state[1][2] = inv_sbox[state[1][2] >> 4][state[1][2] & 0x0F];
    state[1][3] = inv_sbox[state[1][3] >> 4][state[1][3] & 0x0F];
    state[2][0] = inv_sbox[state[2][0] >> 4][state[2][0] & 0x0F];
    state[2][1] = inv_sbox[state[2][1] >> 4][state[2][1] & 0x0F];
    state[2][2] = inv_sbox[state[2][2] >> 4][state[2][2] & 0x0F];
    state[2][3] = inv_sbox[state[2][3] >> 4][state[2][3] & 0x0F];
    state[3][0] = inv_sbox[state[3][0] >> 4][state[3][0] & 0x0F];
    state[3][1] = inv_sbox[state[3][1] >> 4][state[3][1] & 0x0F];
    state[3][2] = inv_sbox[state[3][2] >> 4][state[3][2] & 0x0F];
    state[3][3] = inv_sbox[state[3][3] >> 4][state[3][3] & 0x0F];
}

void ShiftRows(uchar state[][4]){
    int t;
    t = state[1][0];
    state[1][0] = state[1][1];
    state[1][1] = state[1][2];
    state[1][2] = state[1][3];
```

```
    state[1][2] = state[1][3];
    state[1][3] = t;

    t = state[2][0];
    state[2][0] = state[2][2];
    state[2][2] = t;
    t = state[2][1];
    state[2][1] = state[2][3];
    state[2][3] = t;

    t = state[3][0];
    state[3][0] = state[3][3];
    state[3][3] = state[3][2];
    state[3][2] = state[3][1];
    state[3][1] = t;
}

void Inv_ShiftRows(uchar state[][4]){
    int t;
    t = state[1][3];
    state[1][3] = state[1][2];
    state[1][2] = state[1][1];
    state[1][1] = state[1][0];
    state[1][0] = t;

    t = state[2][3];
    state[2][3] = state[2][1];
    state[2][1] = t;
    t = state[2][2];
    state[2][2] = state[2][0];
    state[2][0] = t;

    t = state[3][3];
    state[3][3] = state[3][0];
    state[3][0] = state[3][1];
    state[3][1] = state[3][2];
    state[3][2] = t;
}
```

```
void MixColumns(uchar state [][4]){
    uchar column[4];

    column[0] = state[0][0];
    column[1] = state[1][0];
    column[2] = state[2][0];
    column[3] = state[3][0];
    state[0][0] = gf_mul[column[0]][0];
    state[0][0] ^= gf_mul[column[1]][1];
    state[0][0] ^= column[2];
    state[0][0] ^= column[3];
    state[1][0] = column[0];
    state[1][0] ^= gf_mul[column[1]][0];
    state[1][0] ^= gf_mul[column[2]][1];
    state[1][0] ^= column[3];
    state[2][0] = column[0];
    state[2][0] ^= column[1];
    state[2][0] ^= gf_mul[column[2]][0];
    state[2][0] ^= gf_mul[column[3]][1];
    state[3][0] = gf_mul[column[0]][1];
    state[3][0] ^= column[1];
    state[3][0] ^= column[2];
    state[3][0] ^= gf_mul[column[3]][0];

    column[0] = state[0][1];
    column[1] = state[1][1];
    column[2] = state[2][1];
    column[3] = state[3][1];
    state[0][1] = gf_mul[column[0]][0];
    state[0][1] ^= gf_mul[column[1]][1];
    state[0][1] ^= column[2];
    state[0][1] ^= column[3];
    state[1][1] = column[0];
    state[1][1] ^= gf_mul[column[1]][0];
    state[1][1] ^= gf_mul[column[2]][1];
    state[1][1] ^= column[3];
    state[2][1] = column[0];
    state[2][1] ^= column[1];
```

```
state[2][1] ^= gf_mul[column[2]][0];
state[2][1] ^= gf_mul[column[3]][1];
state[3][1] = gf_mul[column[0]][1];
state[3][1] ^= column[1];
state[3][1] ^= column[2];
state[3][1] ^= gf_mul[column[3]][0];

column[0] = state[0][2];
column[1] = state[1][2];
column[2] = state[2][2];
column[3] = state[3][2];
state[0][2] = gf_mul[column[0]][0];
state[0][2] ^= gf_mul[column[1]][1];
state[0][2] ^= column[2];
state[0][2] ^= column[3];
state[1][2] = column[0];
state[1][2] ^= gf_mul[column[1]][0];
state[1][2] ^= gf_mul[column[2]][1];
state[1][2] ^= column[3];
state[2][2] = column[0];
state[2][2] ^= column[1];
state[2][2] ^= gf_mul[column[2]][0];
state[2][2] ^= gf_mul[column[3]][1];
state[3][2] = gf_mul[column[0]][1];
state[3][2] ^= column[1];
state[3][2] ^= column[2];
state[3][2] ^= gf_mul[column[3]][0];

column[0] = state[0][3];
column[1] = state[1][3];
column[2] = state[2][3];
column[3] = state[3][3];
state[0][3] = gf_mul[column[0]][0];
state[0][3] ^= gf_mul[column[1]][1];
state[0][3] ^= column[2];
state[0][3] ^= column[3];
state[1][3] = column[0];
state[1][3] ^= gf_mul[column[1]][0];
state[1][3] ^= gf_mul[column[2]][1];
```

```
    state[1][3] ^= column[3];
    state[2][3] = column[0];
    state[2][3] ^= column[1];
    state[2][3] ^= gf_mul[column[2]][0];
    state[2][3] ^= gf_mul[column[3]][1];
    state[3][3] = gf_mul[column[0]][1];
    state[3][3] ^= column[1];
    state[3][3] ^= column[2];
    state[3][3] ^= gf_mul[column[3]][0];
}

void Inv_MixColumns(uchar state[][4]){
    int idx;
    uchar column[4],t;

    column[0] = state[0][0];
    column[1] = state[1][0];
    column[2] = state[2][0];
    column[3] = state[3][0];
    state[0][0] = gf_mul[column[0]][5];
    state[0][0] ^= gf_mul[column[1]][3];
    state[0][0] ^= gf_mul[column[2]][4];
    state[0][0] ^= gf_mul[column[3]][2];
    state[1][0] = gf_mul[column[0]][2];
    state[1][0] ^= gf_mul[column[1]][5];
    state[1][0] ^= gf_mul[column[2]][3];
    state[1][0] ^= gf_mul[column[3]][4];
    state[2][0] = gf_mul[column[0]][4];
    state[2][0] ^= gf_mul[column[1]][2];
    state[2][0] ^= gf_mul[column[2]][5];
    state[2][0] ^= gf_mul[column[3]][3];
    state[3][0] = gf_mul[column[0]][3];
    state[3][0] ^= gf_mul[column[1]][4];
    state[3][0] ^= gf_mul[column[2]][2];
    state[3][0] ^= gf_mul[column[3]][5];

    column[0] = state[0][1];
    column[1] = state[1][1];
    column[2] = state[2][1];
```

```
column[3] = state[3][1];
state[0][1] = gf_mul[column[0]][5];
state[0][1] ^= gf_mul[column[1]][3];
state[0][1] ^= gf_mul[column[2]][4];
state[0][1] ^= gf_mul[column[3]][2];
state[1][1] = gf_mul[column[0]][2];
state[1][1] ^= gf_mul[column[1]][5];
state[1][1] ^= gf_mul[column[2]][3];
state[1][1] ^= gf_mul[column[3]][4];
state[2][1] = gf_mul[column[0]][4];
state[2][1] ^= gf_mul[column[1]][2];
state[2][1] ^= gf_mul[column[2]][5];
state[2][1] ^= gf_mul[column[3]][3];
state[3][1] = gf_mul[column[0]][3];
state[3][1] ^= gf_mul[column[1]][4];
state[3][1] ^= gf_mul[column[2]][2];
state[3][1] ^= gf_mul[column[3]][5];

column[0] = state[0][2];
column[1] = state[1][2];
column[2] = state[2][2];
column[3] = state[3][2];
state[0][2] = gf_mul[column[0]][5];
state[0][2] ^= gf_mul[column[1]][3];
state[0][2] ^= gf_mul[column[2]][4];
state[0][2] ^= gf_mul[column[3]][2];
state[1][2] = gf_mul[column[0]][2];
state[1][2] ^= gf_mul[column[1]][5];
state[1][2] ^= gf_mul[column[2]][3];
state[1][2] ^= gf_mul[column[3]][4];
state[2][2] = gf_mul[column[0]][4];
state[2][2] ^= gf_mul[column[1]][2];
state[2][2] ^= gf_mul[column[2]][5];
state[2][2] ^= gf_mul[column[3]][3];
state[3][2] = gf_mul[column[0]][3];
state[3][2] ^= gf_mul[column[1]][4];
state[3][2] ^= gf_mul[column[2]][2];
state[3][2] ^= gf_mul[column[3]][5];
```

```
    column[0] = state[0][3];
    column[1] = state[1][3];
    column[2] = state[2][3];
    column[3] = state[3][3];
    state[0][3] = gf_mul[column[0]][5];
    state[0][3] ^= gf_mul[column[1]][3];
    state[0][3] ^= gf_mul[column[2]][4];
    state[0][3] ^= gf_mul[column[3]][2];
    state[1][3] = gf_mul[column[0]][2];
    state[1][3] ^= gf_mul[column[1]][5];
    state[1][3] ^= gf_mul[column[2]][3];
    state[1][3] ^= gf_mul[column[3]][4];
    state[2][3] = gf_mul[column[0]][4];
    state[2][3] ^= gf_mul[column[1]][2];
    state[2][3] ^= gf_mul[column[2]][5];
    state[2][3] ^= gf_mul[column[3]][3];
    state[3][3] = gf_mul[column[0]][3];
    state[3][3] ^= gf_mul[column[1]][4];
    state[3][3] ^= gf_mul[column[2]][2];
    state[3][3] ^= gf_mul[column[3]][5];
}

uint SubWord(uint word){
    unsigned int result_word;

    result_word = (int)sbox[(word >> 4) & 0x0000000F][word & 0x0000000F];
    result_word += (int)sbox[(word >> 12) & 0x0000000F][(word >> 8) & 0x0000000F] << 8;
    result_word += (int)sbox[(word >> 20) & 0x0000000F][(word >> 16) & 0x0000000F] << 16;
    result_word += (int)sbox[(word >> 28) & 0x0000000F][(word >> 24) & 0x0000000F] << 24;
    return(result_word);
}

void aes_encrypt(uchar input[], uchar output[], uint key[], int keysize){
    uchar temp_state[4][4];
    temp_state[0][0] = input[0];
    temp_state[1][0] = input[1];
    temp_state[2][0] = input[2];
    temp_state[3][0] = input[3];
    temp_state[0][1] = input[4];
```

```
temp_state[1][1] = input[5];
temp_state[2][1] = input[6];
temp_state[3][1] = input[7];
temp_state[0][2] = input[8];
temp_state[1][2] = input[9];
temp_state[2][2] = input[10];
temp_state[3][2] = input[11];
temp_state[0][3] = input[12];
temp_state[1][3] = input[13];
temp_state[2][3] = input[14];
temp_state[3][3] = input[15];

AddRoundKey(temp_state,&key[0]);
SubBytes(temp_state); ShiftRows(temp_state); MixColumns(temp_state); AddRoundKey(temp_state,&key[4]);
SubBytes(temp_state); ShiftRows(temp_state); MixColumns(temp_state); AddRoundKey(temp_state,&key[8]);
SubBytes(temp_state); ShiftRows(temp_state); MixColumns(temp_state); AddRoundKey(temp_state,&key[12]);
SubBytes(temp_state); ShiftRows(temp_state); MixColumns(temp_state); AddRoundKey(temp_state,&key[16]);
SubBytes(temp_state); ShiftRows(temp_state); MixColumns(temp_state); AddRoundKey(temp_state,&key[20]);
SubBytes(temp_state); ShiftRows(temp_state); MixColumns(temp_state); AddRoundKey(temp_state,&key[24]);
SubBytes(temp_state); ShiftRows(temp_state); MixColumns(temp_state); AddRoundKey(temp_state,&key[28]);
SubBytes(temp_state); ShiftRows(temp_state); MixColumns(temp_state); AddRoundKey(temp_state,&key[32]);
SubBytes(temp_state); ShiftRows(temp_state); MixColumns(temp_state); AddRoundKey(temp_state,&key[36]);
if (keysize != 128) {
    SubBytes(temp_state); ShiftRows(temp_state); MixColumns(temp_state); AddRoundKey(temp_state,&key[40]);
    SubBytes(temp_state); ShiftRows(temp_state); MixColumns(temp_state); AddRoundKey(temp_state,&key[44]);
    if (keysize != 192) {
        SubBytes(temp_state); ShiftRows(temp_state); MixColumns(temp_state); AddRoundKey(temp_state,&key[48]);
        SubBytes(temp_state); ShiftRows(temp_state); MixColumns(temp_state); AddRoundKey(temp_state,&key[52]);
        SubBytes(temp_state); ShiftRows(temp_state); AddRoundKey(temp_state,&key[56]);
    }
    else {
        SubBytes(temp_state); ShiftRows(temp_state); AddRoundKey(temp_state,&key[48]);
    }
}
else {
    SubBytes(temp_state); ShiftRows(temp_state); AddRoundKey(temp_state,&key[40]);
}

output[0] = temp_state[0][0];
```

```
    output[1] = temp_state[1][0];
    output[2] = temp_state[2][0];
    output[3] = temp_state[3][0];
    output[4] = temp_state[0][1];
    output[5] = temp_state[1][1];
    output[6] = temp_state[2][1];
    output[7] = temp_state[3][1];
    output[8] = temp_state[0][2];
    output[9] = temp_state[1][2];
    output[10] = temp_state[2][2];
    output[11] = temp_state[3][2];
    output[12] = temp_state[0][3];
    output[13] = temp_state[1][3];
    output[14] = temp_state[2][3];
    output[15] = temp_state[3][3];
}

void aes_decrypt(uchar input[], uchar output[], uint key[], int keysize){
    uchar temp_state[4][4];
    temp_state[0][0] = input[0];
    temp_state[1][0] = input[1];
    temp_state[2][0] = input[2];
    temp_state[3][0] = input[3];
    temp_state[0][1] = input[4];
    temp_state[1][1] = input[5];
    temp_state[2][1] = input[6];
    temp_state[3][1] = input[7];
    temp_state[0][2] = input[8];
    temp_state[1][2] = input[9];
    temp_state[2][2] = input[10];
    temp_state[3][2] = input[11];
    temp_state[0][3] = input[12];
    temp_state[1][3] = input[13];
    temp_state[2][3] = input[14];
    temp_state[3][3] = input[15];

    if (keysize > 128) {
        if (keysize > 192) {
```

```
        AddRoundKey(temp_state,&key[56]);
        Inv_ShiftRows(temp_state);Inv_SubBytes(temp_state);AddRoundKey(temp_state,&key[52]);Inv_MixColumns(temp_state);
        Inv_ShiftRows(temp_state);Inv_SubBytes(temp_state);AddRoundKey(temp_state,&key[48]);Inv_MixColumns(temp_state);
    }
    else {
        AddRoundKey(temp_state,&key[48]);
    }
    Inv_ShiftRows(temp_state);Inv_SubBytes(temp_state);AddRoundKey(temp_state,&key[44]);Inv_MixColumns(temp_state);
    Inv_ShiftRows(temp_state);Inv_SubBytes(temp_state);AddRoundKey(temp_state,&key[40]);Inv_MixColumns(temp_state);
}
else {
    AddRoundKey(temp_state,&key[40]);
}
Inv_ShiftRows(temp_state);Inv_SubBytes(temp_state);AddRoundKey(temp_state,&key[36]);Inv_MixColumns(temp_state);
Inv_ShiftRows(temp_state);Inv_SubBytes(temp_state);AddRoundKey(temp_state,&key[32]);Inv_MixColumns(temp_state);
Inv_ShiftRows(temp_state);Inv_SubBytes(temp_state);AddRoundKey(temp_state,&key[28]);Inv_MixColumns(temp_state);
Inv_ShiftRows(temp_state);Inv_SubBytes(temp_state);AddRoundKey(temp_state,&key[24]);Inv_MixColumns(temp_state);
Inv_ShiftRows(temp_state);Inv_SubBytes(temp_state);AddRoundKey(temp_state,&key[20]);Inv_MixColumns(temp_state);
Inv_ShiftRows(temp_state);Inv_SubBytes(temp_state);AddRoundKey(temp_state,&key[16]);Inv_MixColumns(temp_state);
Inv_ShiftRows(temp_state);Inv_SubBytes(temp_state);AddRoundKey(temp_state,&key[12]);Inv_MixColumns(temp_state);
Inv_ShiftRows(temp_state);Inv_SubBytes(temp_state);AddRoundKey(temp_state,&key[8]);Inv_MixColumns(temp_state);
Inv_ShiftRows(temp_state);Inv_SubBytes(temp_state);AddRoundKey(temp_state,&key[4]);Inv_MixColumns(temp_state);
Inv_ShiftRows(temp_state);Inv_SubBytes(temp_state);AddRoundKey(temp_state,&key[0]);

output[0] = temp_state[0][0];
output[1] = temp_state[1][0];
output[2] = temp_state[2][0];
output[3] = temp_state[3][0];
output[4] = temp_state[0][1];
output[5] = temp_state[1][1];
output[6] = temp_state[2][1];
output[7] = temp_state[3][1];
output[8] = temp_state[0][2];
output[9] = temp_state[1][2];
output[10] = temp_state[2][2];
output[11] = temp_state[3][2];
output[12] = temp_state[0][3];
output[13] = temp_state[1][3];
output[14] = temp_state[2][3];
```

```
    output[15] = temp_state[3][3];
}

void XTS_AES_block(uchar tweak[], uchar ciphertext[], uint key1[],uint key2[], int i){
    aes_encrypt(tweak,ciphertext_tweak,key_schedule2,256);

    jj=j;
    while(jj!=0){
            LCheckBit=0;
        for (idxx=0; idxx < 16; idxx++) {
            MCheckBit = (ciphertext_tweak[idxx] >> 7);
            ciphertext_tweak[idxx] = (ciphertext_tweak[idxx] << 1)& 0xff;

            if(LCheckBit == 1){
                ciphertext_tweak[idxx]=ciphertext_tweak[idxx]^0x01;
            }

            LCheckBit = MCheckBit;
            if (MCheckBit == 1 && idxx == 15){
                ciphertext_tweak[0]=ciphertext_tweak[0]^0x87;
            }
        }
        jj--;
    }

    for (idx=0; idx < 16; idx++){
        T[idx]=ciphertext_tweak[idx];
        PP[idx]= T[idx]^ciphertext[idx];
    }

    aes_decrypt(PP,CC,key_schedule1,256);

    for (idx=0; idx < 16; idx++){
```

```
        plaintexts[i][idx]= T[idx]^CC[idx];
    }
}

#define KE_ROTWORD(z) ( ((z) << 8) | ((z) >> 24) )

void KeyExpansion(uchar key[], uint w[], int keysize){
   int Nb=4,Nr,Nk,i;
   uint temp,Rcon[]={0x01000000,0x02000000,0x04000000,0x08000000,0x10000000,0x20000000,
                     0x40000000,0x80000000,0x1b000000,0x36000000,0x6c000000,0xd8000000,
                     0xab000000,0x4d000000,0x9a000000};
   switch (keysize) {
      case 128: Nr = 10; Nk = 4; break;
      case 192: Nr = 12; Nk = 6; break;
      case 256: Nr = 14; Nk = 8; break;
      default: return;
   }

   for (i=0; i < Nk; ++i) {
      w[i] = ((key[4 * i]) << 24) | ((key[4 * i + 1]) << 16) |
             ((key[4 * i + 2]) << 8) | ((key[4 * i + 3]));
   }

   for (i = Nk; i < Nb * (Nr+1); ++i) {
      temp = w[i - 1];
      if ((i % Nk) == 0)
         temp = SubWord(KE_ROTWORD(temp)) ^ Rcon[(i-1)/Nk];
      else if (Nk > 6 && (i % Nk) == 4)
         temp = SubWord(temp);
      w[i] = w[i-Nk] ^ temp;
   }
}

/*~~~~~~~~~~~~~~~~~~~~~~~~~MAIN FUNCTION~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
int main(int argc, char **argv) {
    clock_t begin, end;
    double time_spent;
    begin = clock();
    int nprocs, myrank;
```

```
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    int i, j, k, s=15, l=0;
    static uchar global_input[TOTAL_NUM_OF_ITEMS/2];uchar local_input[numi];
        uchar line[3], local_tweak[numi];
    static uchar global_tweak[TOTAL_NUM_OF_TWEAKS/2];

    unsigned char key1[32] = {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,
                              0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f,0x10,0x11,
                          0x12,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1a,
                          0x1b,0x1c,0x1d,0x1e,0x1f},
                  key2[32] = {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,
                          0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f,0x10,0x11,
                          0x12,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1a,
                          0x1b,0x1c,0x1d,0x1e,0x1f};

    FILE * fi, * ft, * fo;

/*~~~~~~~~~~~~~~~~~PROCESSOR ZERO INITIAL ACTIONS~~~~~~~~~~~~~~~~*/
    if (myrank == 0) {
            fi = fopen("output.txt", "rt");
        if (fi == NULL){
            printf("could not open the input file. \n\n");
            exit(1);
        }
        for (i=0; i<TOTAL_NUM_OF_ITEMS/2; i++){
                    if(fgets(line, 3, fi) != NULL){
                sscanf(line, "%02x", &global_input[i]);
            }
            else {
                global_input[i]=0;
            }
        }

        ft = fopen("tweak.txt", "rt");
        if (ft == NULL){
```

```
        printf("could not open the input tweak file. \n\n");
        exit(1);
    }
    for (i=0; i<TOTAL_NUM_OF_TWEAKS/2; i++){
        if(fgets(line, 3, ft) != NULL){
        sscanf(line, "%02x", &global_tweak[i]);
        } else {
        global_tweak[i]=0;
        }
    }

    fclose(fi);
    fclose(ft);

    //expanding key1 and key2
    KeyExpansion(key1,key_schedule1,256);
    KeyExpansion(key2,key_schedule2,256);
}

MPI_Scatter(global_input, numi, MPI_UNSIGNED_CHAR,&local_input, numi,
MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);

MPI_Barrier(MPI_COMM_WORLD);

MPI_Scatter(global_tweak, numi, MPI_UNSIGNED_CHAR, &local_tweak, numi,
MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);

MPI_Barrier(MPI_COMM_WORLD);

//putting input plain-texts into 2D array
k = 0;
for (i=0; i<numb; i++) {
    for(j=0;j<16;j++){
        ciphertexts[i][j] = local_input[k];
        k++;
    }
}
```

```
//putting tweaks into 2D array
k = 0;
for (i=0; i<numb; i++) {
    for(j=0;j<16;j++){
        tweaks[i][j] = local_tweak[k];
        k++;
    }
}

//processing each block
j = numb * myrank;
for (i =0; i <numb; i++) {
    j = j+1;
    for(k=0;k<16;k++){
        if (i != numb) {
                    ciphertext[k]= ciphertexts[i][k];
        }else {
            if(ciphertexts[i][k]!='\0'){
                s=k;
                ciphertext[k]=ciphertexts[i][k];
            } else{
                ciphertext[k]=plaintexts[i-1][k];
            }
        }
        tweak[k]=tweaks[i][k];
    }

if(s == 15){
    XTS_AES_block(tweak, ciphertext, key_schedule1, key_schedule2,i);
}else {
    while(l<s || l ==s){
        plaintexts[i][l]=plaintexts[i-1][l];
        l++;
    }
    for(l=s+1;l<16;l++){
        plaintexts[i][l]='\0';
    }
    XTS_AES_block(tweak, ciphertext,key_schedule1, key_schedule2,i-1);
}
```

```
}

k=0;
for(i=0; i<numb; i++){
    for(j=0; j<16; j++){
        local_input[k] = plaintexts[i][j];
        k++;
    }
}

MPI_Barrier(MPI_COMM_WORLD);

MPI_Gather(&local_input, numi, MPI_UNSIGNED_CHAR, global_input, numi,
        MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);

if (myrank == 0) {
        fo = fopen("temp.txt","wt");
    for(i=0; i<TOTAL_NUM_OF_ITEMS/2; i++){
        fprintf(fo,"%02x",global_input[i]);
    }
    fclose(fo);

    fo = fopen("p1.txt","rt");
    fi = fopen("temp.txt","rt");

    int ch1, ch2;
    ch1=getc(fi); ch2=getc(fo);

    while((ch1!=EOF) && (ch2!= EOF) && (ch1==ch2)){
        ch1 =getc(fi);
        ch2 = getc(fo);
    }

    if(ch1==ch2) printf("\n Identical \n");
    else if(ch1 != ch2) printf("\nNot Identical \n");

    fclose(fi);
    fclose(fo);
```

```
        end = clock();
        time_spent=(double)(end-begin)/CLOCKS_PER_SEC;
        printf("Total time = %f\n\n",time_spent);
    }

    MPI_Finalize();
        return 0;
}
```

[C] XTS_AES_Serial.c

```
#include <stdio.h>
#include <time.h>

#define uchar unsigned char
#define uint unsigned long
#define numb 64

FILE * FileP, * FileC, * FileO;
uchar plaintext[16], plaintexts[numb][16], line[3],second_last_plaintext[16];
uchar ciphertext[16], ciphertexts[numb][16];

int LCheckBit, MCheckBit, j=0, jj, k;
uchar tweak[16],tweaks[numb][16];
uchar ciphertext_tweak[16],ciphertext_tweaks[numb][16], T[16],temp[16], PP[16],CC[16];
uchar key1[32]={0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,
                0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f,0x10,0x11,
                0x12,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1a,
                0x1b,0x1c,0x1d,0x1e,0x1f};
uchar key2[32]={0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,
                0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f,0x10,0x11,
                0x12,0x13,0x14,0x15,0x16,0x17,0x18,0x19,0x1a,
                0x1b,0x1c,0x1d,0x1e,0x1f};
unsigned int key_schedule1[60],key_schedule2[60],idx,idxx,first=1;

uchar sbox[16][16] = {
   0x63,0x7C,0x77,0x7B,0xF2,0x6B,0x6F,0xC5,0x30,0x01,0x67,0x2B,0xFE,0xD7,0xAB,0x76,
   0xCA,0x82,0xC9,0x7D,0xFA,0x59,0x47,0xF0,0xAD,0xD4,0xA2,0xAF,0x9C,0xA4,0x72,0xC0,
   0xB7,0xFD,0x93,0x26,0x36,0x3F,0xF7,0xCC,0x34,0xA5,0xE5,0xF1,0x71,0xD8,0x31,0x15,
   0x04,0xC7,0x23,0xC3,0x18,0x96,0x05,0x9A,0x07,0x12,0x80,0xE2,0xEB,0x27,0xB2,0x75,
   0x09,0x83,0x2C,0x1A,0x1B,0x6E,0x5A,0xA0,0x52,0x3B,0xD6,0xB3,0x29,0xE3,0x2F,0x84,
   0x53,0xD1,0x00,0xED,0x20,0xFC,0xB1,0x5B,0x6A,0xCB,0xBE,0x39,0x4A,0x4C,0x58,0xCF,
   0xD0,0xEF,0xAA,0xFB,0x43,0x4D,0x33,0x85,0x45,0xF9,0x02,0x7F,0x50,0x3C,0x9F,0xA8,
   0x51,0xA3,0x40,0x8F,0x92,0x9D,0x38,0xF5,0xBC,0xB6,0xDA,0x21,0x10,0xFF,0xF3,0xD2,
   0xCD,0x0C,0x13,0xEC,0x5F,0x97,0x44,0x17,0xC4,0xA7,0x7E,0x3D,0x64,0x5D,0x19,0x73,
   0x60,0x81,0x4F,0xDC,0x22,0x2A,0x90,0x88,0x46,0xEE,0xB8,0x14,0xDE,0x5E,0x0B,0xDB,
   0xE0,0x32,0x3A,0x0A,0x49,0x06,0x24,0x5C,0xC2,0xD3,0xAC,0x62,0x91,0x95,0xE4,0x79,
   0xE7,0xC8,0x37,0x6D,0x8D,0xD5,0x4E,0xA9,0x6C,0x56,0xF4,0xEA,0x65,0x7A,0xAE,0x08,
   0xBA,0x78,0x25,0x2E,0x1C,0xA6,0xB4,0xC6,0xE8,0xDD,0x74,0x1F,0x4B,0xBD,0x8B,0x8A,
   0x70,0x3E,0xB5,0x66,0x48,0x03,0xF6,0x0E,0x61,0x35,0x57,0xB9,0x86,0xC1,0x1D,0x9E,
```

```
    0xE1,0xF8,0x98,0x11,0x69,0xD9,0x8E,0x94,0x9B,0x1E,0x87,0xE9,0xCE,0x55,0x28,0xDF,
    0x8C,0xA1,0x89,0x0D,0xBF,0xE6,0x42,0x68,0x41,0x99,0x2D,0x0F,0xB0,0x54,0xBB,0x16
};
uchar gf_mul[256][6] = {
    {0x00,0x00,0x00,0x00,0x00,0x00},{0x02,0x03,0x09,0x0b,0x0d,0x0e},
    {0x04,0x06,0x12,0x16,0x1a,0x1c},{0x06,0x05,0x1b,0x1d,0x17,0x12},
    {0x08,0x0c,0x24,0x2c,0x34,0x38},{0x0a,0x0f,0x2d,0x27,0x39,0x36},
    {0x0c,0x0a,0x36,0x3a,0x2e,0x24},{0x0e,0x09,0x3f,0x31,0x23,0x2a},
    {0x10,0x18,0x48,0x58,0x68,0x70},{0x12,0x1b,0x41,0x53,0x65,0x7e},
    {0x14,0x1e,0x5a,0x4e,0x72,0x6c},{0x16,0x1d,0x53,0x45,0x7f,0x62},
    {0x18,0x14,0x6c,0x74,0x5c,0x48},{0x1a,0x17,0x65,0x7f,0x51,0x46},
    {0x1c,0x12,0x7e,0x62,0x46,0x54},{0x1e,0x11,0x77,0x69,0x4b,0x5a},
    {0x20,0x30,0x90,0xb0,0xd0,0xe0},{0x22,0x33,0x99,0xbb,0xdd,0xee},
    {0x24,0x36,0x82,0xa6,0xca,0xfc},{0x26,0x35,0x8b,0xad,0xc7,0xf2},
    {0x28,0x3c,0xb4,0x9c,0xe4,0xd8},{0x2a,0x3f,0xbd,0x97,0xe9,0xd6},
    {0x2c,0x3a,0xa6,0x8a,0xfe,0xc4},{0x2e,0x39,0xaf,0x81,0xf3,0xca},
    {0x30,0x28,0xd8,0xe8,0xb8,0x90},{0x32,0x2b,0xd1,0xe3,0xb5,0x9e},
    {0x34,0x2e,0xca,0xfe,0xa2,0x8c},{0x36,0x2d,0xc3,0xf5,0xaf,0x82},
    {0x38,0x24,0xfc,0xc4,0x8c,0xa8},{0x3a,0x27,0xf5,0xcf,0x81,0xa6},
    {0x3c,0x22,0xee,0xd2,0x96,0xb4},{0x3e,0x21,0xe7,0xd9,0x9b,0xba},
    {0x40,0x60,0x3b,0x7b,0xbb,0xdb},{0x42,0x63,0x32,0x70,0xb6,0xd5},
    {0x44,0x66,0x29,0x6d,0xa1,0xc7},{0x46,0x65,0x20,0x66,0xac,0xc9},
    {0x48,0x6c,0x1f,0x57,0x8f,0xe3},{0x4a,0x6f,0x16,0x5c,0x82,0xed},
    {0x4c,0x6a,0x0d,0x41,0x95,0xff},{0x4e,0x69,0x04,0x4a,0x98,0xf1},
    {0x50,0x78,0x73,0x23,0xd3,0xab},{0x52,0x7b,0x7a,0x28,0xde,0xa5},
    {0x54,0x7e,0x61,0x35,0xc9,0xb7},{0x56,0x7d,0x68,0x3e,0xc4,0xb9},
    {0x58,0x74,0x57,0x0f,0xe7,0x93},{0x5a,0x77,0x5e,0x04,0xea,0x9d},
    {0x5c,0x72,0x45,0x19,0xfd,0x8f},{0x5e,0x71,0x4c,0x12,0xf0,0x81},
    {0x60,0x50,0xab,0xcb,0x6b,0x3b},{0x62,0x53,0xa2,0xc0,0x66,0x35},
    {0x64,0x56,0xb9,0xdd,0x71,0x27},{0x66,0x55,0xb0,0xd6,0x7c,0x29},
    {0x68,0x5c,0x8f,0xe7,0x5f,0x03},{0x6a,0x5f,0x86,0xec,0x52,0x0d},
    {0x6c,0x5a,0x9d,0xf1,0x45,0x1f},{0x6e,0x59,0x94,0xfa,0x48,0x11},
    {0x70,0x48,0xe3,0x93,0x03,0x4b},{0x72,0x4b,0xea,0x98,0x0e,0x45},
    {0x74,0x4e,0xf1,0x85,0x19,0x57},{0x76,0x4d,0xf8,0x8e,0x14,0x59},
    {0x78,0x44,0xc7,0xbf,0x37,0x73},{0x7a,0x47,0xce,0xb4,0x3a,0x7d},
    {0x7c,0x42,0xd5,0xa9,0x2d,0x6f},{0x7e,0x41,0xdc,0xa2,0x20,0x61},
    {0x80,0xc0,0x76,0xf6,0x6d,0xad},{0x82,0xc3,0x7f,0xfd,0x60,0xa3},
    {0x84,0xc6,0x64,0xe0,0x77,0xb1},{0x86,0xc5,0x6d,0xeb,0x7a,0xbf},
    {0x88,0xcc,0x52,0xda,0x59,0x95},{0x8a,0xcf,0x5b,0xd1,0x54,0x9b},
```

```
{0x8c,0xca,0x40,0xcc,0x43,0x89},{0x8e,0xc9,0x49,0xc7,0x4e,0x87},
{0x90,0xd8,0x3e,0xae,0x05,0xdd},{0x92,0xdb,0x37,0xa5,0x08,0xd3},
{0x94,0xde,0x2c,0xb8,0x1f,0xc1},{0x96,0xdd,0x25,0xb3,0x12,0xcf},
{0x98,0xd4,0x1a,0x82,0x31,0xe5},{0x9a,0xd7,0x13,0x89,0x3c,0xeb},
{0x9c,0xd2,0x08,0x94,0x2b,0xf9},{0x9e,0xd1,0x01,0x9f,0x26,0xf7},
{0xa0,0xf0,0xe6,0x46,0xbd,0x4d},{0xa2,0xf3,0xef,0x4d,0xb0,0x43},
{0xa4,0xf6,0xf4,0x50,0xa7,0x51},{0xa6,0xf5,0xfd,0x5b,0xaa,0x5f},
{0xa8,0xfc,0xc2,0x6a,0x89,0x75},{0xaa,0xff,0xcb,0x61,0x84,0x7b},
{0xac,0xfa,0xd0,0x7c,0x93,0x69},{0xae,0xf9,0xd9,0x77,0x9e,0x67},
{0xb0,0xe8,0xae,0x1e,0xd5,0x3d},{0xb2,0xeb,0xa7,0x15,0xd8,0x33},
{0xb4,0xee,0xbc,0x08,0xcf,0x21},{0xb6,0xed,0xb5,0x03,0xc2,0x2f},
{0xb8,0xe4,0x8a,0x32,0xe1,0x05},{0xba,0xe7,0x83,0x39,0xec,0x0b},
{0xbc,0xe2,0x98,0x24,0xfb,0x19},{0xbe,0xe1,0x91,0x2f,0xf6,0x17},
{0xc0,0xa0,0x4d,0x8d,0xd6,0x76},{0xc2,0xa3,0x44,0x86,0xdb,0x78},
{0xc4,0xa6,0x5f,0x9b,0xcc,0x6a},{0xc6,0xa5,0x56,0x90,0xc1,0x64},
{0xc8,0xac,0x69,0xa1,0xe2,0x4e},{0xca,0xaf,0x60,0xaa,0xef,0x40},
{0xcc,0xaa,0x7b,0xb7,0xf8,0x52},{0xce,0xa9,0x72,0xbc,0xf5,0x5c},
{0xd0,0xb8,0x05,0xd5,0xbe,0x06},{0xd2,0xbb,0x0c,0xde,0xb3,0x08},
{0xd4,0xbe,0x17,0xc3,0xa4,0x1a},{0xd6,0xbd,0x1e,0xc8,0xa9,0x14},
{0xd8,0xb4,0x21,0xf9,0x8a,0x3e},{0xda,0xb7,0x28,0xf2,0x87,0x30},
{0xdc,0xb2,0x33,0xef,0x90,0x22},{0xde,0xb1,0x3a,0xe4,0x9d,0x2c},
{0xe0,0x90,0xdd,0x3d,0x06,0x96},{0xe2,0x93,0xd4,0x36,0x0b,0x98},
{0xe4,0x96,0xcf,0x2b,0x1c,0x8a},{0xe6,0x95,0xc6,0x20,0x11,0x84},
{0xe8,0x9c,0xf9,0x11,0x32,0xae},{0xea,0x9f,0xf0,0x1a,0x3f,0xa0},
{0xec,0x9a,0xeb,0x07,0x28,0xb2},{0xee,0x99,0xe2,0x0c,0x25,0xbc},
{0xf0,0x88,0x95,0x65,0x6e,0xe6},{0xf2,0x8b,0x9c,0x6e,0x63,0xe8},
{0xf4,0x8e,0x87,0x73,0x74,0xfa},{0xf6,0x8d,0x8e,0x78,0x79,0xf4},
{0xf8,0x84,0xb1,0x49,0x5a,0xde},{0xfa,0x87,0xb8,0x42,0x57,0xd0},
{0xfc,0x82,0xa3,0x5f,0x40,0xc2},{0xfe,0x81,0xaa,0x54,0x4d,0xcc},
{0x1b,0x9b,0xec,0xf7,0xda,0x41},{0x19,0x98,0xe5,0xfc,0xd7,0x4f},
{0x1f,0x9d,0xfe,0xe1,0xc0,0x5d},{0x1d,0x9e,0xf7,0xea,0xcd,0x53},
{0x13,0x97,0xc8,0xdb,0xee,0x79},{0x11,0x94,0xc1,0xd0,0xe3,0x77},
{0x17,0x91,0xda,0xcd,0xf4,0x65},{0x15,0x92,0xd3,0xc6,0xf9,0x6b},
{0x0b,0x83,0xa4,0xaf,0xb2,0x31},{0x09,0x80,0xad,0xa4,0xbf,0x3f},
{0x0f,0x85,0xb6,0xb9,0xa8,0x2d},{0x0d,0x86,0xbf,0xb2,0xa5,0x23},
{0x03,0x8f,0x80,0x83,0x86,0x09},{0x01,0x8c,0x89,0x88,0x8b,0x07},
{0x07,0x89,0x92,0x95,0x9c,0x15},{0x05,0x8a,0x9b,0x9e,0x91,0x1b},
{0x3b,0xab,0x7c,0x47,0x0a,0xa1},{0x39,0xa8,0x75,0x4c,0x07,0xaf},
{0x3f,0xad,0x6e,0x51,0x10,0xbd},{0x3d,0xae,0x67,0x5a,0x1d,0xb3},
```

```
{0x33,0xa7,0x58,0x6b,0x3e,0x99},{0x31,0xa4,0x51,0x60,0x33,0x97},
{0x37,0xa1,0x4a,0x7d,0x24,0x85},{0x35,0xa2,0x43,0x76,0x29,0x8b},
{0x2b,0xb3,0x34,0x1f,0x62,0xd1},{0x29,0xb0,0x3d,0x14,0x6f,0xdf},
{0x2f,0xb5,0x26,0x09,0x78,0xcd},{0x2d,0xb6,0x2f,0x02,0x75,0xc3},
{0x23,0xbf,0x10,0x33,0x56,0xe9},{0x21,0xbc,0x19,0x38,0x5b,0xe7},
{0x27,0xb9,0x02,0x25,0x4c,0xf5},{0x25,0xba,0x0b,0x2e,0x41,0xfb},
{0x5b,0xfb,0xd7,0x8c,0x61,0x9a},{0x59,0xf8,0xde,0x87,0x6c,0x94},
{0x5f,0xfd,0xc5,0x9a,0x7b,0x86},{0x5d,0xfe,0xcc,0x91,0x76,0x88},
{0x53,0xf7,0xf3,0xa0,0x55,0xa2},{0x51,0xf4,0xfa,0xab,0x58,0xac},
{0x57,0xf1,0xe1,0xb6,0x4f,0xbe},{0x55,0xf2,0xe8,0xbd,0x42,0xb0},
{0x4b,0xe3,0x9f,0xd4,0x09,0xea},{0x49,0xe0,0x96,0xdf,0x04,0xe4},
{0x4f,0xe5,0x8d,0xc2,0x13,0xf6},{0x4d,0xe6,0x84,0xc9,0x1e,0xf8},
{0x43,0xef,0xbb,0xf8,0x3d,0xd2},{0x41,0xec,0xb2,0xf3,0x30,0xdc},
{0x47,0xe9,0xa9,0xee,0x27,0xce},{0x45,0xea,0xa0,0xe5,0x2a,0xc0},
{0x7b,0xcb,0x47,0x3c,0xb1,0x7a},{0x79,0xc8,0x4e,0x37,0xbc,0x74},
{0x7f,0xcd,0x55,0x2a,0xab,0x66},{0x7d,0xce,0x5c,0x21,0xa6,0x68},
{0x73,0xc7,0x63,0x10,0x85,0x42},{0x71,0xc4,0x6a,0x1b,0x88,0x4c},
{0x77,0xc1,0x71,0x06,0x9f,0x5e},{0x75,0xc2,0x78,0x0d,0x92,0x50},
{0x6b,0xd3,0x0f,0x64,0xd9,0x0a},{0x69,0xd0,0x06,0x6f,0xd4,0x04},
{0x6f,0xd5,0x1d,0x72,0xc3,0x16},{0x6d,0xd6,0x14,0x79,0xce,0x18},
{0x63,0xdf,0x2b,0x48,0xed,0x32},{0x61,0xdc,0x22,0x43,0xe0,0x3c},
{0x67,0xd9,0x39,0x5e,0xf7,0x2e},{0x65,0xda,0x30,0x55,0xfa,0x20},
{0x9b,0x5b,0x9a,0x01,0xb7,0xec},{0x99,0x58,0x93,0x0a,0xba,0xe2},
{0x9f,0x5d,0x88,0x17,0xad,0xf0},{0x9d,0x5e,0x81,0x1c,0xa0,0xfe},
{0x93,0x57,0xbe,0x2d,0x83,0xd4},{0x91,0x54,0xb7,0x26,0x8e,0xda},
{0x97,0x51,0xac,0x3b,0x99,0xc8},{0x95,0x52,0xa5,0x30,0x94,0xc6},
{0x8b,0x43,0xd2,0x59,0xdf,0x9c},{0x89,0x40,0xdb,0x52,0xd2,0x92},
{0x8f,0x45,0xc0,0x4f,0xc5,0x80},{0x8d,0x46,0xc9,0x44,0xc8,0x8e},
{0x83,0x4f,0xf6,0x75,0xeb,0xa4},{0x81,0x4c,0xff,0x7e,0xe6,0xaa},
{0x87,0x49,0xe4,0x63,0xf1,0xb8},{0x85,0x4a,0xed,0x68,0xfc,0xb6},
{0xbb,0x6b,0x0a,0xb1,0x67,0x0c},{0xb9,0x68,0x03,0xba,0x6a,0x02},
{0xbf,0x6d,0x18,0xa7,0x7d,0x10},{0xbd,0x6e,0x11,0xac,0x70,0x1e},
{0xb3,0x67,0x2e,0x9d,0x53,0x34},{0xb1,0x64,0x27,0x96,0x5e,0x3a},
{0xb7,0x61,0x3c,0x8b,0x49,0x28},{0xb5,0x62,0x35,0x80,0x44,0x26},
{0xab,0x73,0x42,0xe9,0x0f,0x7c},{0xa9,0x70,0x4b,0xe2,0x02,0x72},
{0xaf,0x75,0x50,0xff,0x15,0x60},{0xad,0x76,0x59,0xf4,0x18,0x6e},
{0xa3,0x7f,0x66,0xc5,0x3b,0x44},{0xa1,0x7c,0x6f,0xce,0x36,0x4a},
{0xa7,0x79,0x74,0xd3,0x21,0x58},{0xa5,0x7a,0x7d,0xd8,0x2c,0x56},
{0xdb,0x3b,0xa1,0x7a,0x0c,0x37},{0xd9,0x38,0xa8,0x71,0x01,0x39},
```

```
    {0xdf,0x3d,0xb3,0x6c,0x16,0x2b},{0xdd,0x3e,0xba,0x67,0x1b,0x25},
    {0xd3,0x37,0x85,0x56,0x38,0x0f},{0xd1,0x34,0x8c,0x5d,0x35,0x01},
    {0xd7,0x31,0x97,0x40,0x22,0x13},{0xd5,0x32,0x9e,0x4b,0x2f,0x1d},
    {0xcb,0x23,0xe9,0x22,0x64,0x47},{0xc9,0x20,0xe0,0x29,0x69,0x49},
    {0xcf,0x25,0xfb,0x34,0x7e,0x5b},{0xcd,0x26,0xf2,0x3f,0x73,0x55},
    {0xc3,0x2f,0xcd,0x0e,0x50,0x7f},{0xc1,0x2c,0xc4,0x05,0x5d,0x71},
    {0xc7,0x29,0xdf,0x18,0x4a,0x63},{0xc5,0x2a,0xd6,0x13,0x47,0x6d},
    {0xfb,0x0b,0x31,0xca,0xdc,0xd7},{0xf9,0x08,0x38,0xc1,0xd1,0xd9},
    {0xff,0x0d,0x23,0xdc,0xc6,0xcb},{0xfd,0x0e,0x2a,0xd7,0xcb,0xc5},
    {0xf3,0x07,0x15,0xe6,0xe8,0xef},{0xf1,0x04,0x1c,0xed,0xe5,0xe1},
    {0xf7,0x01,0x07,0xf0,0xf2,0xf3},{0xf5,0x02,0x0e,0xfb,0xff,0xfd},
    {0xeb,0x13,0x79,0x92,0xb4,0xa7},{0xe9,0x10,0x70,0x99,0xb9,0xa9},
    {0xef,0x15,0x6b,0x84,0xae,0xbb},{0xed,0x16,0x62,0x8f,0xa3,0xb5},
    {0xe3,0x1f,0x5d,0xbe,0x80,0x9f},{0xe1,0x1c,0x54,0xb5,0x8d,0x91},
    {0xe7,0x19,0x4f,0xa8,0x9a,0x83},{0xe5,0x1a,0x46,0xa3,0x97,0x8d}
};

void AddRoundKey(uchar state[][4], uint w[]){
    uchar sub_key[4];

    sub_key[0] = w[0] >> 24;
    sub_key[1] = w[0] >> 16;
    sub_key[2] = w[0] >> 8;
    sub_key[3] = w[0];
    state[0][0] ^= sub_key[0];
    state[1][0] ^= sub_key[1];
    state[2][0] ^= sub_key[2];
    state[3][0] ^= sub_key[3];

    sub_key[0] = w[1] >> 24;
    sub_key[1] = w[1] >> 16;
    sub_key[2] = w[1] >> 8;
    sub_key[3] = w[1];
    state[0][1] ^= sub_key[0];
    state[1][1] ^= sub_key[1];
    state[2][1] ^= sub_key[2];
    state[3][1] ^= sub_key[3];

    sub_key[0] = w[2] >> 24;
```

```
    sub_key[1] = w[2] >> 16;
    sub_key[2] = w[2] >> 8;
    sub_key[3] = w[2];
    state[0][2] ^= sub_key[0];
    state[1][2] ^= sub_key[1];
    state[2][2] ^= sub_key[2];
    state[3][2] ^= sub_key[3];

    sub_key[0] = w[3] >> 24;
    sub_key[1] = w[3] >> 16;
    sub_key[2] = w[3] >> 8;
    sub_key[3] = w[3];
    state[0][3] ^= sub_key[0];
    state[1][3] ^= sub_key[1];
    state[2][3] ^= sub_key[2];
    state[3][3] ^= sub_key[3];
}

void SubBytes(uchar state[][4]){
    state[0][0] = sbox[state[0][0] >> 4][state[0][0] & 0x0F];
    state[0][1] = sbox[state[0][1] >> 4][state[0][1] & 0x0F];
    state[0][2] = sbox[state[0][2] >> 4][state[0][2] & 0x0F];
    state[0][3] = sbox[state[0][3] >> 4][state[0][3] & 0x0F];
    state[1][0] = sbox[state[1][0] >> 4][state[1][0] & 0x0F];
    state[1][1] = sbox[state[1][1] >> 4][state[1][1] & 0x0F];
    state[1][2] = sbox[state[1][2] >> 4][state[1][2] & 0x0F];
    state[1][3] = sbox[state[1][3] >> 4][state[1][3] & 0x0F];
    state[2][0] = sbox[state[2][0] >> 4][state[2][0] & 0x0F];
    state[2][1] = sbox[state[2][1] >> 4][state[2][1] & 0x0F];
    state[2][2] = sbox[state[2][2] >> 4][state[2][2] & 0x0F];
    state[2][3] = sbox[state[2][3] >> 4][state[2][3] & 0x0F];
    state[3][0] = sbox[state[3][0] >> 4][state[3][0] & 0x0F];
    state[3][1] = sbox[state[3][1] >> 4][state[3][1] & 0x0F];
    state[3][2] = sbox[state[3][2] >> 4][state[3][2] & 0x0F];
    state[3][3] = sbox[state[3][3] >> 4][state[3][3] & 0x0F];
}

void ShiftRows(uchar state[][4]){
    int t;
```

```
    state[1][0] = state[1][1];
    state[1][1] = state[1][2];
    state[1][2] = state[1][3];
    state[1][3] = t;

    t = state[2][0];
    state[2][0] = state[2][2];
    state[2][2] = t;
    t = state[2][1];
    state[2][1] = state[2][3];
    state[2][3] = t;

    t = state[3][0];
    state[3][0] = state[3][3];
    state[3][3] = state[3][2];
    state[3][2] = state[3][1];
    state[3][1] = t;
}

void MixColumns(uchar state[][4]){
    uchar column[4];
    column[0] = state[0][0];
    column[1] = state[1][0];
    column[2] = state[2][0];
    column[3] = state[3][0];
    state[0][0] = gf_mul[column[0]][0];
    state[0][0] ^= gf_mul[column[1]][1];
    state[0][0] ^= column[2];
    state[0][0] ^= column[3];
    state[1][0] = column[0];
    state[1][0] ^= gf_mul[column[1]][0];
    state[1][0] ^= gf_mul[column[2]][1];
    state[1][0] ^= column[3];
    state[2][0] = column[0];
    state[2][0] ^= column[1];
    state[2][0] ^= gf_mul[column[2]][0];
    state[2][0] ^= gf_mul[column[3]][1];
    state[3][0] = gf_mul[column[0]][1];
    state[3][0] ^= column[1];
```

```
state[3][0] ^= column[2];
state[3][0] ^= gf_mul[column[3]][0];

column[0] = state[0][1];
column[1] = state[1][1];
column[2] = state[2][1];
column[3] = state[3][1];
state[0][1] = gf_mul[column[0]][0];
state[0][1] ^= gf_mul[column[1]][1];
state[0][1] ^= column[2];
state[0][1] ^= column[3];
state[1][1] = column[0];
state[1][1] ^= gf_mul[column[1]][0];
state[1][1] ^= gf_mul[column[2]][1];
state[1][1] ^= column[3];
state[2][1] = column[0];
state[2][1] ^= column[1];
state[2][1] ^= gf_mul[column[2]][0];
state[2][1] ^= gf_mul[column[3]][1];
state[3][1] = gf_mul[column[0]][1];
state[3][1] ^= column[1];
state[3][1] ^= column[2];
state[3][1] ^= gf_mul[column[3]][0];

column[0] = state[0][2];
column[1] = state[1][2];
column[2] = state[2][2];
column[3] = state[3][2];
state[0][2] = gf_mul[column[0]][0];
state[0][2] ^= gf_mul[column[1]][1];
state[0][2] ^= column[2];
state[0][2] ^= column[3];
state[1][2] = column[0];
state[1][2] ^= gf_mul[column[1]][0];
state[1][2] ^= gf_mul[column[2]][1];
state[1][2] ^= column[3];
state[2][2] = column[0];
state[2][2] ^= column[1];
state[2][2] ^= gf_mul[column[2]][0];
```

```
    state[2][2] ^= gf_mul[column[3]][1];
    state[3][2] = gf_mul[column[0]][1];
    state[3][2] ^= column[1];
    state[3][2] ^= column[2];
    state[3][2] ^= gf_mul[column[3]][0];

    column[0] = state[0][3];
    column[1] = state[1][3];
    column[2] = state[2][3];
    column[3] = state[3][3];
    state[0][3] = gf_mul[column[0]][0];
    state[0][3] ^= gf_mul[column[1]][1];
    state[0][3] ^= column[2];
    state[0][3] ^= column[3];
    state[1][3] = column[0];
    state[1][3] ^= gf_mul[column[1]][0];
    state[1][3] ^= gf_mul[column[2]][1];
    state[1][3] ^= column[3];
    state[2][3] = column[0];
    state[2][3] ^= column[1];
    state[2][3] ^= gf_mul[column[2]][0];
    state[2][3] ^= gf_mul[column[3]][1];
    state[3][3] = gf_mul[column[0]][1];
    state[3][3] ^= column[1];
    state[3][3] ^= column[2];
    state[3][3] ^= gf_mul[column[3]][0];
}

uint SubWord(uint word){
    unsigned int result_word;

    result_word = (int)sbox[(word >> 4) & 0x0000000F][word & 0x0000000F];
    result_word += (int)sbox[(word >> 12) & 0x0000000F][(word >> 8) & 0x0000000F] << 8;
    result_word += (int)sbox[(word >> 20) & 0x0000000F][(word >> 16) & 0x0000000F] << 16
    result_word += (int)sbox[(word >> 28) & 0x0000000F][(word >> 24) & 0x0000000F] << 24
    return(result_word);
}

#define KE ROTWORD(z) ( ((z) << 8) | ((z) >> 24) )
```

```
void KeyExpansion(uchar key[], uint w[], int keysize){
   int Nb=4,Nr,Nk,i;
   uint temp,Rcon[]={0x01000000,0x02000000,0x04000000,0x08000000,0x10000000,0x20000000,
                     0x40000000,0x80000000,0x1b000000,0x36000000,0x6c000000,0xd8000000,
                     0xab000000,0x4d000000,0x9a000000};
   switch (keysize) {
      case 128: Nr = 10; Nk = 4; break;
      case 192: Nr = 12; Nk = 6; break;
      case 256: Nr = 14; Nk = 8; break;
      default: return;
   }

   for (i=0; i < Nk; ++i) {
      w[i] = ((key[4 * i]) << 24) | ((key[4 * i + 1]) << 16) |
             ((key[4 * i + 2]) << 8) | ((key[4 * i + 3]));
   }

   for (i = Nk; i < Nb * (Nr+1); ++i) {
      temp = w[i - 1];
      if ((i % Nk) == 0)
         temp = SubWord(KE_ROTWORD(temp)) ^ Rcon[(i-1)/Nk];
      else if (Nk > 6 && (i % Nk) == 4)
         temp = SubWord(temp);
      w[i] = w[i-Nk] ^ temp;
   }
}

void aes_encrypt(uchar input[], uchar output[], uint key[], inputt keysize){
   uchar temp_state[4][4];

   temp_state[0][0] = input[0];
   temp_state[1][0] = input[1];
   temp_state[2][0] = input[2];
   temp_state[3][0] = input[3];
   temp_state[0][1] = input[4];
   temp_state[1][1] = input[5];
   temp_state[2][1] = input[6];
   temp_state[3][1] = input[7];
   temp_state[0][2] = input[8];
```

```
state[1][2] = in[9];
state[2][2] = in[10];
state[3][2] = in[11];
state[0][3] = in[12];
state[1][3] = in[13];
state[2][3] = in[14];
state[3][3] = in[15];

AddRoundKey(state,&key[0]);
SubBytes(state); ShiftRows(state); MixColumns(state); AddRoundKey(state,&key[4]);
SubBytes(state); ShiftRows(state); MixColumns(state); AddRoundKey(state,&key[8]);
SubBytes(state); ShiftRows(state); MixColumns(state); AddRoundKey(state,&key[12]);
SubBytes(state); ShiftRows(state); MixColumns(state); AddRoundKey(state,&key[16]);
SubBytes(state); ShiftRows(state); MixColumns(state); AddRoundKey(state,&key[20]);
SubBytes(state); ShiftRows(state); MixColumns(state); AddRoundKey(state,&key[24]);
SubBytes(state); ShiftRows(state); MixColumns(state); AddRoundKey(state,&key[28]);
SubBytes(state); ShiftRows(state); MixColumns(state); AddRoundKey(state,&key[32]);
SubBytes(state); ShiftRows(state); MixColumns(state); AddRoundKey(state,&key[36]);
if (keysize != 128) {
    SubBytes(state); ShiftRows(state); MixColumns(state); AddRoundKey(state,&key[40]);
    SubBytes(state); ShiftRows(state); MixColumns(state); AddRoundKey(state,&key[44]);
    if (keysize != 192) {
        SubBytes(state); ShiftRows(state); MixColumns(state); AddRoundKey(state,&key[48]);
        SubBytes(state); ShiftRows(state); MixColumns(state); AddRoundKey(state,&key[52]);
        SubBytes(state); ShiftRows(state); AddRoundKey(state,&key[56]);
    }
    else {
        SubBytes(state); ShiftRows(state); AddRoundKey(state,&key[48]);
    }
}
else {
    SubBytes(state); ShiftRows(state); AddRoundKey(state,&key[40]);
}

out[0] = state[0][0];
out[1] = state[1][0];
out[2] = state[2][0];
out[3] = state[3][0];
out[4] = state[0][1];
```

```
    out[5] = state[1][1];
    out[6] = state[2][1];
    out[7] = state[3][1];
    out[8] = state[0][2];
    out[9] = state[1][2];
    out[10] = state[2][2];
    out[11] = state[3][2];
    out[12] = state[0][3];
    out[13] = state[1][3];
    out[14] = state[2][3];
    out[15] = state[3][3];
}

void XTS_AES_block(uchar tweak[], uchar plaintext[], uint key1[],uint key2[], int i){

    aes_encrypt(tweak,ciphertext_tweak,key_schedule2,256);

    jj=j;
    while(jj!=0){
        LCheckBit=0;

        for (idxx=0; idxx < 16; idxx++) {
            MCheckBit = (ciphertext_tweak[idxx] >> 7);
            ciphertext_tweak[idxx] = (ciphertext_tweak[idxx] << 1)& 0xff;
            if(LCheckBit == 1){
                ciphertext_tweak[idxx]=ciphertext_tweak[idxx]^0x01;
            }
            LCheckBit = MCheckBit;
            if (MCheckBit == 1 && idxx == 15){
            ciphertext_tweak[0]=ciphertext_tweak[0]^0x87;
        }}
        jj--;
    }

    for (idx=0; idx < 16; idx++){
        T[idx]=ciphertext_tweak[idx];
        PP[idx]= T[idx]^plaintext[idx];
    }
```

```
    if(i+1==numb-1 && first ==1){
        for(idx=0; idx < 16; idx++){
            temp[idx]=ciphertext_tweak[idx];
        }
            first=0;
    }

    aes_encrypt(PP,CC,key_schedule1,256);

    for (idx=0; idx < 16; idx++){
        ciphertexts[i][idx]= T[idx]^CC[idx];
    }
}

int main(){
    clock_t begin, end;
    double time_spent;
    begin = clock();

    int i, s, l=0, x=0,y=0;

    FILE* fp;
    FILE* ft;
    fp = fopen ("p.txt", "rt");
    ft = fopen ("tweak.txt", "rt");

    for(x=0;x<numb;x++){
        for(y=0;y<16;y++){
                if(fgets(line, 3, fp) != NULL){
                    sscanf (line, "%02x", &plaintexts[x][y]);
                }
                else {plaintexts[x][y] = 0;}

        }
   }
   for(x=0;x<numb;x++){
        for(y=0;y<16;y++){
                if(fgets(line, 3, ft) != NULL){
                    sscanf (line, "%02x", &tweaks[x][y]);
```

```
            }
            else {tweaks[x][y] = 0;}

        }
    }

    fclose(fp);
    fclose(ft);

    KeyExpansion(key1,key_schedule1,256);
    KeyExpansion(key2,key_schedule2,256);

    FileP = fopen("plain.txt","wt");
    FileC = fopen("cipher.txt","wt");
    FileO = fopen("output.txt","wt");

    for (i =0; i <= numb-1; i++) {
     for(k=0;k<16;k++){
            if(plaintexts[i][k]!='\0'){
                s=k;
                plaintext[k]=plaintexts[i][k];
            }
            else{
                plaintext[k]=ciphertexts[i-1][k];
            }
            tweak[k]=tweaks[i][k];
        }

  if(s == 15) {
    XTS_AES_block(tweak, plaintext,key_schedule1, key_schedule2,i);
  } else {
        for (idx=0; idx < 16; idx++){
            second_last_plaintext[idx]= plaintexts[i-1][idx];
        }

        while(l<s || l ==s){
            ciphertexts[i][l]=ciphertexts[i-1][l];
            l++;
        }
```

```
        for(l=s+1;l<16;l++){
            ciphertexts[i][l]='\0';
        }
        XTS_AES_block(tweak, plaintext,key_schedule1, key_schedule2,i-1);
    }
    j++;
    }

  for(i=0;i<numb;i++){
    for(l=0;l<16;l++){
            fprintf(FileC,"%02x",ciphertexts[i][l]);
        }
    }

    fclose(FileC);
    fclose(FileP);
    fclose(FileO);

    end = clock();
    time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
    printf("Total execution time = %f\n\n", time_spent);

   getchar();
   return 0;
}
```