Efficient Implementation of Advanced Encryption Standard

Project Report submitted in partial fulfillment of the requirement for the degree of

Bachelor of Technology

in

Computer Science & Engineering

under the Supervision of

Mr. Amit Kumar Singh

by

Shubham Dwivedee
Enrollment No.111339

to

Jaypee University of Information Technology

Waknaghat, Solan – 173234, Himachal Pradesh

Certificate

This is to certify that project report entitled **"Efficient Implementation of Advanced Encryption Standard"** submitted by **Shubham Dwivedee** in partial fulfillment for the award of degree of Bachelor of Technology in Computer Science & Engineering to Jaypee University of Information Technology, Waknaghat, Solan has been carried out under my supervision.

This work has not been submitted partially or fully to any other University or Institute for the award of this or any other degree or diploma.

Date:

Supervisor's Name

Designation

Acknowledgement

I wish to express my profound gratitude and indebtedness to **Mr. Amit Kumar Singh**, for their continuous support, inspiring guidance, constructive criticism and valuable suggestion throughout the project work. Their guidance has helped me at all times of my research and writing of this report.

I would also like to thank **Prof. Dr. Satya Prakash Ghrera** for sharing his vast expanse of knowledge in guiding me with the correct books and sparing his valuable time and helped me in striving to move forward to this point. Without their valuable inputs, I wouldn't have been able to incrementally work well and go ahead with the project.

Last but not the least, my sincere thanks to all my friends who have patiently extended all sorts of help for accomplishing this undertaking.

Date: **Name of the Student**

Table of Contents

List of Figures

S. No. **Title** **Page No.**

List of Tables

S. No. **Title** **Page No.**

Abstract

In today's world most of the communication is done using electronic media. Data Security plays a vital role in such communication. Hence, there is a need to protect data from malicious attacks. This can be achieved by using Cryptographic techniques. The earlier encryption algorithm is Data Encryption Standard (DES) which has several loopholes such as small key size and sensible to brute force attack etc. and it cannot provide high level, efficient and exportable security. These loopholes overcome by a new algorithm called as Advanced Encryption Standard (AES). It was created by two Belgian cryptographers, Vincent Rijmen and Joan Daemen, replacing the old Data Encryption Standard (DES). The Federal Information Processing Standard 197 used a standardized version of the algorithm called Rijndael for the Advanced Encryption Standard. AES was successful because it was easy to implement and could run in a reasonable amount of time on a regular computer.

In this project work, the plain text of 128 bits is given as input to encryption block in which encryption of data is made and the cipher text of 128 bits is throughout as output. The key length of 128bits, 192bits or 256bits is used in process of encryption. The AES algorithm is a block cipher that uses the same binary key for both encryption and decryption of data blocks. Hence it is called a symmetric key cryptography. The rounds in decryption are exact inverse of encryption. There are four rounds in encrypt ions viz. Sub Bytes, ShiftRows, MixColumns and AddRoundKey. Similarly for Decryption we have InvSubBytes, InvShiffilows, InvMixColumns and InvAddRoundKey. The number of times operation performed is depend on key length i.e. for 128bis we have 10 rounds. Since operations in AES are difficult, there exists no attack better than key exhaustion to read an encrypted message. Ultimately, anyone can use AES encryption methods, and it is free for public or private, commercial or non-commercial use. The simplest version encrypts and decrypts each 128 -bit block individually. It gives better security than DES versions and also better throughput.

Chapter 1

Introduction

In today's world most of the communication is done using electronic media. Data security plays a vital role in communication via internet. Hence, there is a need to protect data from malicious attacks. This can be achieved by Cryptography. The earlier encryption algorithm is Data Encryption Standard (DES) which has several loopholes like small key size that makes it prone to brute force attacks, etc. It fails to provide high level, efficient and exportable security. These loopholes were overcome by a new algorithm called Advanced Encryption Standard (AES).

In this project work, the plain text of 128 bits is given as input to encryption block in which encryption of data is made and the cipher text of 128 bits is throughout as output. The key length of 128bits, 192bits or 256bits is used in process of encryption. The AES algorithm is a block cipher that uses the same binary key for both encryption and decryption of data blocks.

1.1. Purpose

Due to the advancements in the Internet technology, huge digital data are transmitted over the public network. As the public network is open to all, protection of these data is a vital issue. Thus for protecting these data from the unauthorized people, Cryptography has come up as a solution which plays a vital role in information security system against various attacks. Advanced Encryption Standard is the current standard for symmetric key cryptography and is considered very much secure due to it

1.2. Motivation

The Advanced Encryption Standard, in the following referenced as AES is the winner of the contest, held in 1997 by the US Government, after the Data Encryption Standard (DES) was found too weak. Fifteen candidates were accepted in 1998 and based on public comments the pool was reduced to five finalists in 1999. In October 2000, one of these five algorithms was selected as the forthcoming standard: a slightly modified version of the Rijndael. The Rijndael, whose name is based on the names of its two Belgian inventors Joan Daemen and Vincent Rijmen, is a Block cipher, which means that it works on fixed -length group of bits, which are called Blocks. It takes an input block of a certain size usually 128 bits, and produces a corresponding output block of the same size. The transformation requires a second input, which is the secret key. It is important to know that the secret key can be of any size (depending on the cipher used) and that AES uses three different key sizes: 128, 192 and 256 bits.

1.3. Overview

Advanced Encryption Standard (AES) is a symmetric key cryptography and it has block cipher with a fixed block size of 128 bit and a variable key length i.e. it may be 128, 192 or 256 bits. The different transformations operate on the intermediate results, called state. The state is a rectangular array of bytes and since the block size is 128 bits, which is 16 bytes, the rectangular array is of dimensions 4x4. (In the Rijndael version with variable block size, the row size is fixed to four and the number of columns varies. The number of columns is the block size divided by 32 and denoted Nb). The cipher key is similarly pictured as a rectangular array with four rows. The number of columns of the cipher key is equal to the key length divided by 32.

AES uses a variable number of rounds, which are fixed: A key of size 128 has 10 rounds. A key of size 192 has 12 rounds. A key of size 256 has 14 rounds. An algorithm starts with a random number, in which the key and data encrypted with it are scrambled though four mathematical operation processes. The key that is used to encrypt the number must also be used to decrypt it. For encryption, each rounds has four operations SubBytes, ShiftRows, MixColumns and AddRoundKey respectively and for decryption it use inverse of these function.

AES does not use a Feistel structure but processes the entire data block in parallel during each round using substitutions and permutation. The key that is provided as input is expanded into an array of forty-four 32 –bit words. Four distinct words (128 bits) serve as a round key for each round. Four different stages are used, one of permutation and three of substitution.

- SubstituteBytes: Uses a table, referred to as an S -box, to perform a byte by byte substitution of the block
- ShiftRows: A simple permutation that is performed row by row
- MixColumns: A substitution that alters each byte in a column as function of all of the bytes in the column

- AddRoundkey: A simple bitwise XOR of the current block with a portion of the expanded key

The structure is quite simple. For both encryption and decryption, the cipher begins with an Add Round Key stage, followed by nine rounds that each includes all four stages, followed by a tenth round of three stages.

Only the Add Round Key stage makes use of the key. For this reason, the cipher begins and ends with an Add Round Key stage. Any other stage, applied at the beginning or end, is reversible without knowledge of the key and so would add no security.

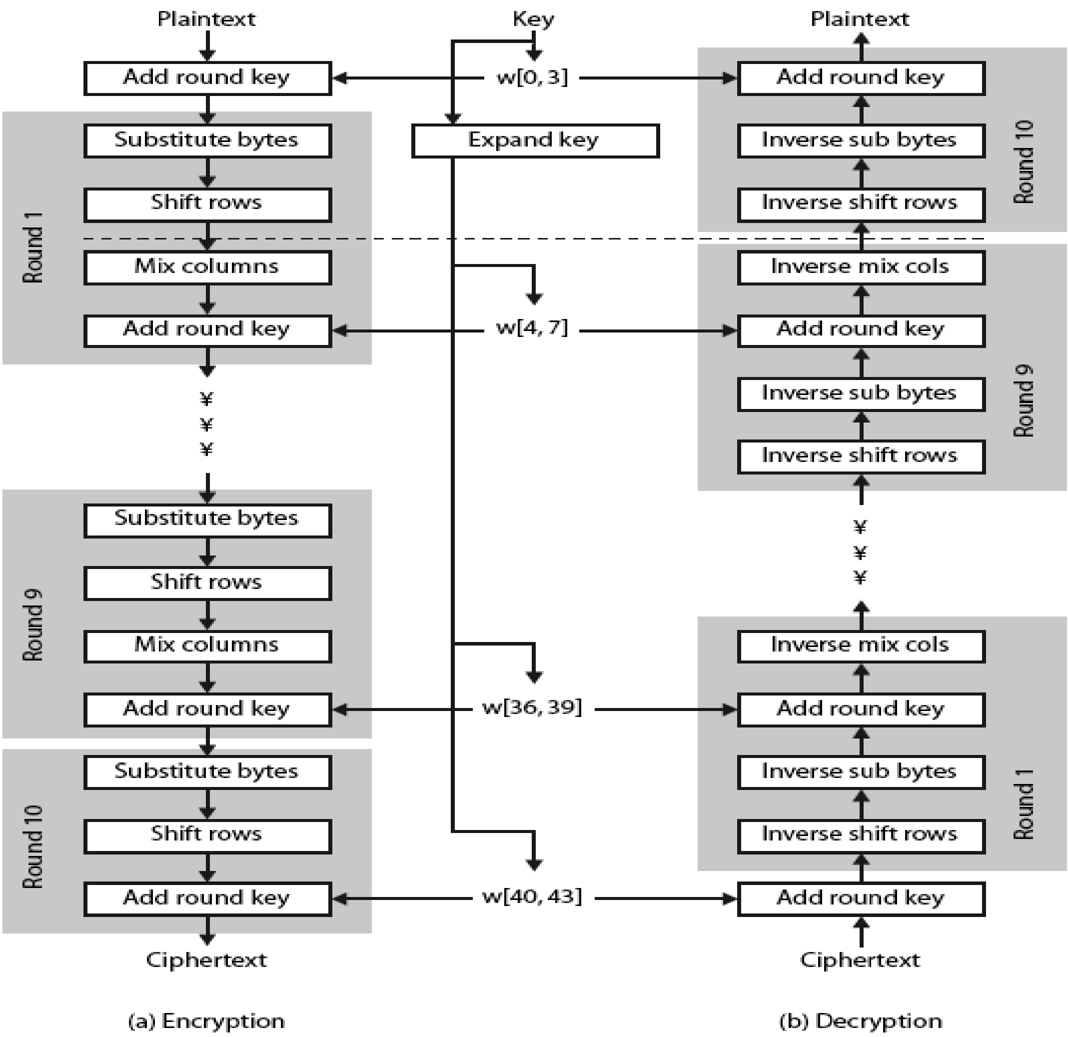

Figure 1: AES Structure [1]

The Add Round Key stage by itself would not be formidable. The other three stages together scramble the bits, but by themselves, they would provide no security because they do not use the key. We can view the cipher as alternating operations of XOR encryption (Add Round Key) of a block, followed by scrambling of the block (the other three stages), and followed by XOR encryption, and so on. This scheme is both efficient and highly secure. Each stage is easily reversible. For the Substitute Byte. Shift Row, and Mix Columns stages, an inverse function is used in the decryption algorithm. For the Add Round Key stage, the inverse is achieved by X0Ring the same round key to the block, using the result that A (I) B (I) B = A.

As with most block ciphers, the decryption algorithm makes use of the expanded key in reverse order. However, the decryption algorithm is not identical to the encryption algorithm. This is a consequence of the particular structure of AES. Once it is established that all four stages are reversible, it is easy to verify that decryption does recover the plaintext.

1.4. Background

On January 2, 1997 the National Institute of Standards and Technology (NIST) held a contest for a new encryption standard. The previous standard, DES, was no longer adequate for security. It had been the standard since November 23, 1976. Computing power had increased a lot since then and the algorithm was no longer considered safe. The earlier ciphers can be broken with ease on modern computation systems. In 1998 DES was cracked in less than three days by a specially made computer called the DES cracker. The DES cracker was created by the Electronic Frontier Foundation for less than $250,000 and won the RSA DES Challenge II-2. It was also far too slow in software as it was developed for mid-1970's hardware and does not produce efficient software code. Triple DES on the other hand, has three times as many rounds as DES and is correspondingly slower. As well as this, the 64 bit block size of triple DES and DES is not very efficient and is questionable when it comes to security Current alternatives to a new encryption standard were Triple DES (3DES) and International Data Encryption Algorithm (IDEA). The problem was IDEA and 3DES were too slow and IDEA was not free to implement due to patents. NIST wanted a free and easy to implement algorithm that would provide good security. Additionally they wanted the algorithm to be efficient and flexible.

What was required was a brand new encryption algorithm. One that would be resistant to all known attacks. The National Institute of Standards and Technology (NIST) wanted to help in the creation of a new standard. However, because of the controversy that went with the DES algorithm, and the years of some branches of the U.S. government trying everything they could to hinder deployment of secure cryptography this was likely to raise strong skepticism. The problem was that NIST did actually want to help create a new excellent encryption standard but they couldn't get involved directly. Unfortunately they were really the only ones with the technical reputation and resources to the lead the effort.

Table 1: First Round Qualifiers [3]

ALGORITHM NAME	SUBMITTER
CAST-256	Entrust Technologies, Inc.
CRYPTON	Future Systems, Inc.
DEAL	Richard Outerbridge, Lars Knudsen
DFC	CNRS - Centre National pour la Recherche Scientifique - Ecole Normale Superieure
E2	NTT - Nippon Telegraph and Telephone Corporation
FROG	TecApro Internacional S.A.
HPC	Rich Schroeppel
LOKI97	Lawrie Brown, Josef Pieprzyk, Jennifer Seberry
MAGENTA	Deutsche Telekom AG
MARS	IBM
RC 6	RSA Laboratories
Rijndael	Joaen Daemen, Vincent Rijmen
SAFER+	Cylink Corporation
Serpent	Ross Anderson, Eli Biham, Lars Knudsen
Twofish	Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, Niels Ferguson

Instead of designing or helping to design a cipher, what they did instead was to set up a contest in which anyone in the world could take part. The contest was announced on the 2nd January 1997 and the idea was to develop a new encryption algorithm that would be used for protecting sensitive, non-classified, U.S. government information. The ciphers had to meet a lot of requirements and the whole design had to be fully documented (unlike the DES cipher). Once the candidate algorithms had been submitted, several years of scrutiny in the form of cryptographic conferences took place. In the first round of the competition 15 algorithms were accepted and this was narrowed to 5 in the second round. The fifteen algorithms are shown in table below of which the 5 that were selected are shown in bold. The algorithms were tested for efficiency and security both by some of the world's best publicly renowned cryptographers and NIST itself.

After holding the contest for three years, NIST chose an algorithm created by two Belgian computer scientists, Vincent Rijmen and Joan Daemen. On November 26, 2001 the Federal Information Processing Standards Publication 197 announced a standardized form of the Rijndael algorithm as the new standard for encryption. This standard was called Advanced Encryption Standard and is currently the standard for encryption.

1.5. Definitions

Cryptography: Cryptography is the science of secret codes, enabling the confidentiality of communication through an insecure channel. It protects against unauthorized parties by preventing unauthorized alteration of use. Generally speaking, it uses a cryptographic system to transform a plaintext into a cipher text most of the time using a key. It has different Encryption and Decryption algorithms to do so.

Cipher Text: This is the scrambled message produced as output from Encryption algorithm. It depends on the plaintext and the secret key. For a given message, two different keys will produce two different cipher texts.

Encryption: Encryption is the process of converting data, in plain text format into a meaningless cipher text by means of a suitable algorithm. The algorithm takes secret key and plain text as input and produces cipher text.

Decryption: Decryption is converting the meaningless cipher text into the original information using decryption algorithms. The decryption algorithm is inverse of encryption algorithm. This takes key and cipher text as input and produces original plain text.

Symmetric key cryptography: Symmetric cryptography uses the same secret (private) key to encrypt and decrypt its data. It requires that the secret key be known by the party encrypting the data and the party decrypting the data.

Asymmetric key cryptography: Asymmetric uses both a public and private key. This allows for distribution of your public key to anyone with which they can encrypt the data they want to send securely and then it can only be decoded by the person having the private key.

1.6. AES vs DES

There is a huge, important difference between these two encryption and decryption algorithms, Data Encryption Standard (DES) and the Advanced Encryption Standard (AES): AES is secure while DES is not. The federal government developed DES encryption algorithms more than 30 years ago to provide cryptographic security for all government communications. The idea was to ensure government systems all used the same, secure standard to facilitate interconnectivity. DES served as the cornerstone of government cryptography for more than two decades, but in 1999 researchers broke the algorithm's 56-bit key using a distributed computer system. AES data encryption is a more mathematically efficient and elegant cryptographic algorithm, but its main strength rests in the key length options. The time required to crack an encryption algorithm is directly related to the length of the key used to secure the communication. AES allows you to choose a 128-bit, 192-bit or 256-bit key, making it exponentially stronger than the 56-bit key of DES.

Data Encryption Standard is a rather old way of encrypting data so that the information could not be read by other people who might be intercepting traffic. DES is rather quite old and has since been replaced by a newer and better Advanced Encryption Standard. The replacement was done due to the inherent weaknesses in DES that allowed the encryption to be broken using certain methods of attack. Common applications of AES, as of the moment, are still impervious to any type of cracking techniques, which makes it a good choice even for top secret information.

The inherent weakness in DES is caused by a couple of things that are already addressed in AES. The first is the very short 56 bit encryption key. The key is like a password that is necessary in order to decrypt the information. A 56 bit has a maximum of 256 combinations, which might seem like a lot but is rather easy for a computer to do a brute force attack on. AES can use a 128, 192, or 256 bit encryption key with 2^128, 2^192, 2^256 combinations respectively. The longer encryption keys make it much harder to break given that the system has no other weaknesses.

Another problem is the small block size used by DES, which is set at 64 bits. In comparison, AES uses a block size that is twice as long at 128 bits. In simple terms, the block size determines how much information you can send before you start having identical blocks, which leak information. People can intercept these blocks and use read the leaked information. For DES with 64 bits, the maximum amount of data that can be transferred with a single encryption key is 32GB; at this point another key needs to be used. With AES, it is at 256 exabytes or 256 billion gigabytes. It is probably safe to say that you can use a single AES encryption key for any application.

In terms of structure, DES uses the Feistel network which divides the block into two halves before going through the encryption steps. AES on the other hand, uses permutation-substitution, which involves a series of substitution and permutation steps to create the encrypted block. Summing up we can say that:

- DES is really old while AES is relatively new
- DES is breakable while AES is still unbreakable
- DES uses a much smaller key size compared to AES
- DES uses a smaller block size compared to AES
- DES uses a balanced Feistel structure while AES uses substitution-permutation

1.7. AES vs 3DES

Advance Encryption Standard (AES) and Triple DES (TDES or 3DES) are commonly used block ciphers. Whether you choose AES or 3DES depend on your needs. DES was developed in 1977 and it was carefully designed to work better in hardware than software. DES performs lots of bit manipulation in substitution and permutation boxes in each of 16 rounds. Even though it seems large but according to today’s computing power it is not sufficient and vulnerable to brute force attack. Therefore, DES could not keep up with advancement in technology and it is no longer appropriate for security. Because DES was widely used at that time, the quick solution was to introduce 3DES which is secure enough for most purposes today. 3DES is a construction of applying DES three times in sequence. 3DES with three different keys (K1, K2 and K3) has effective key length is 168 bits (The use of three distinct key is recommended of 3DES.). Another variation is called two-key (K1 and K3 is same) 3DES reduces the effective key size to 112 bits which is less secure. Two-key 3DES is widely used in electronic payments industry. 3DES takes three times as much CPU power than compare with its predecessor which is significant performance hit. AES outperforms 3DES both in software and in hardware.

AES (Advanced Encryption Standard) and 3DES, or also known as Triple DES (Data Encryption Standard) are two of the current standards in data encryption. While AES is a totally new encryption that uses the substitution-permutation network, 3DES is just an adaptation to the older DES encryption that relied on the balanced Feistel network. Basically, 3DES is just DES applied three times to the information that is being encrypted.

AES uses three common encryption key lengths, 128, 192, and 256 bits. When it comes to 3DES the encryption key is still limited to 56 bits as dictated by the DES standard. But since it is applied three times, the implementer can choose to have 3 discrete 56 bit keys, or 2 identical and 1 discrete, or even three identical keys. This means that 3DES can have encryption key lengths of 168, 112, or 56 bit encryption key lengths respectively. But due to certain vulnerabilities when reapplying the same encryption thrice, using 168 bits has a reduced security equivalent to 112 bits and using 112 bits has a reduced security equivalent to 80 bits.

3DES also uses the same block length of 64 bits, half the size that of AES at 128 bits. Using AES provides additional insurance that it is harder to sniff leaked data from identical blocks. When using 3DES, the user needs to switch encryption keys every 32GB of data transfer to minimize the possibility of leaks; identical to when using the standard DES encryption.

Lastly, repeating the same process three times does take some time. With all things held constant, AES is much faster compared to 3DES. This line gets blurred when you include software, hardware, and the complexity of hardware design to the mix. So if you have 3DES accelerated hardware, migrating to AES implemented by software alone may result in slower processing times. In this aspect, there is not better solution than to test each one and measure their speed. But when it comes to security, AES is the sure winner as it is still considered unbreakable in practical use. Summing up:

- 3DES uses identical encryption to DES while AES uses a totally different
- 3DES has shorter and weaker encryption keys compared to AES
- 3DES uses repeating encryption keys while AES does not

- 3DES also uses a shorter block length compared to AES
- 3DES encryption takes longer than AES encryption

1.8. AES vs RSA

RSA is one of the most successful, asymmetric encryption systems today. Originally discovered 1973 by the British intelligence agency GCHQ, it received the classification “top secret”. Its civil rediscovery is owned to the cryptologists Rivest, Shamir and Adleman, who discovered it during an attempt to break another cryptographic problem. As opposed to traditional, symmetric encryption systems, RSA works with two different keys: A “public” key, and a “private” one. Both work complementary to each other, a message encrypted with one of them can only be decrypted by its counterpart. Since the private key can’t be calculated from the public key, the latter is generally made available to the public. Those properties enable asymmetric cryptosystems to be used in a wide array of functions, such as digital signatures. In the process of signing a document, a fingerprint, encrypted with RSA, is appended to the file, and enables the receiver to verify both the sender and the integrity of the document.

The security of RSA itself is mainly based on the mathematical problem of integer factorization. A message that is about to be encrypted is treated as one large number. When encrypting the message, it is raised to the power of the key, and divided with remainder by a fixed product of two primes. By repeating the process with the other key, the plaintext can be retrieved back. The best, currently known method to break the encryption requires factorizing the product used in the division. Currently, it is not possible to calculate these factors for numbers greater than 768 bits. None the less, modern cryptosystems use a minimum key length of 3072 bits.

As first publicly accessible, from the NSA for the classification "top secret" approved cipher, the Advanced Encryption Standard (AES) is one of the most frequently used and most secure encryption algorithms available today. Its story of success started 1997, when the National Institute of Standards and Technology NIST announced the search for a successor to the aging encryption standard DES. An algorithm named "Rijndael", developed by the Belgian cryptographists Daemen and Rijmen, excelled in security as well as in performance and flexibility. It came out on top of several competitors, and was officially announced as the new encryption standard AES in 2001. The algorithm is based on several substitutions, permutations and linear transformations, each executed on data blocks of 16 byte – therefore the term blockcipher. Those operations are repeated several times, called “rounds”. During each round, a unique roundkey is calculated out of the encryption key, and incorporated in the calculations. Based on this block structure of AES, the change of a single bit either in the key, or in the plaintext block results in a completely different ciphertext block – a clear advantage over traditional stream ciphers. The difference between AES-128, AES-192 and AES-256 finally is the length of the key: 128, 192 or 256 bit – all drastic improvements compared to the 56 bit key of DES. By way of illustration: Cracking a 128 bit AES key with a state-of-the-art supercomputer would take longer than the presumed age of the universe. And Boxcryptor even uses 256 bit keys! As of today, no practicable attack against AES exists. Therefore, AES remains the preferred encryption standard for governments, banks and high security systems around the world.

They're not really directly comparable. The number commonly bandied about is 2048-bit RSA is about equivalent to 128-bit AES. But that number shouldn't be relied on without understanding the caveats. Currently the most effective way of breaking AES crypto (and any other unbroken symmetric cipher, for that matter) is brute-force. You simply try every possibility until you reach the correct result. This means that it is possible, and well within today's technology, to encrypt data that (assuming no better attack is ever found), can never be broken, ever, by anyone. Simply use enough bits in your key such that there isn't enough energy in the universe to try enough candidate keys. The numbers are smaller than you'd think: Indeed, with AES, 128-bit is secure against modern technology, 256 is secure against any likely future technology, and 512 is probably secure against even never-imagined hypothetical alien technology.

Symmetric encryption, if not broken, doesn't leave you with a math problem to solve. The numbers are truly and literally scrambled, and the system is devised such the brute-force is by far the most efficient solution. Breaking RSA, on the other hand, is not so hard. Instead of brute-forcing the keys, you factor the modulus into primes and derive the keys yourself. This is dramatically simpler to do. It's a math problem, and we can do math. Specifically, the speed at which primes can be factored is increasing faster than the speed at which symmetric keys can be brute-forced. And that's with today's technology. But going forward, assuming quantum computers can be improved such that qbit operations are a cheap as bit operations (which many people thinks is fairly close; this century at most, possibly decades), then no matter how large you make your RSA key, breaking the key is as fast as encrypting.

Summing up one would say that equivalent security of RSA key length versus AES key length changes over time. Every so often, you have to increase your RSA key size relative to your AES key size to account for technological advances. And even then, it's an estimate at best. And while a 256-bit symmetric key should be secure for hundreds, thousands, or perhaps hundreds of thousands of years, no RSA key of any length should be assumed to be secure more than a few dozen years out, since RSA is expected to be completely and utterly broken by Shor's algorithm.

Table 2: Comparison between DES, AES and RSA [3]

S.NO.	FACTOR	DES	AES	RSA
1	Developed	1977	2000	1978
2	Key Length Value	56 bit	128, 192 and 256 bits	>1024 bits
3	Type of Algorithm	Symmetric	Symmetric	Asymmetric
4	Encryption Ratio	Low	High	High
5	Security Attacks	Inadequate	Highly Secured	Timing attack
6	Simulation Speed	Fast	Fast	Fast
7	Scalability	Scalable algorithm	No scalability occurs	No scalability occurs
8	Power Consumption	Low	Low	High
9	Hardware and Software Implementations	Better in hardware than in software	Faster and efficient	Not very efficient

1.9. Organization of the Report

This report document comprises of five chapters. The Chapter 1 gives the overview to AES algorithm, basic definitions of terms that are used in this report and purpose of project and also gives the motivation behind implementing this project. Chapter 2 gives the details of requirements for implementing the project. It gives hardware, software and user requirements and the performance parameters taken into consideration. Chapter 3 gives the research analysis regarding AES algorithm also the basics about multithreading and parallel execution. Chapter 4 gives the details of each modules used in this project and some implementation details. Chapter 5 gives conclusion, limitations and further enhancement to the project. References section provide source detail where we get information. Appendix contains snapshots of the project code execution.

Chapter 2

System Requirement Specification

The following are the system requirements:

2.1. Hardware Requirements

- 512MB RAM or above
- X86 or above processor
- 2MB Secondary memory or above

2.2. Software Requirements

- Operating System: LINUX, Windows
- Language used: Java
- Editor: Eclipse IDE

2.3. Functional Requirements

The functional requirements for the implementation are as follows:

2.3.1. Input Specification

- An input file/string type variable should contain some data. That can be used as plain text for encryption
- Secret key used for encryption should of l28bits, 192bits or 256bits

2.3.2. Output Specification

- The second party should know secret key that used for encryption.
- After providing secret key as input, it displays the original plain text.

2.4. Performance Parameters

The performance of AES algorithm can be measured by considering following parameters:

2.4.1. Time Taken

The time taken for encryption as well as decryption of a given plain text is calculated by using system clock time: The system clock is recorded twice i.e. before and after the execution of the encryption module and their difference yields the time taken for encryption. The same procedure is followed to calculate decryption time, just that decryption module is invoked instead.

2.4.2. Throughput

In computer technology, throughput is the amount of work that a computer can do in a given time period. Throughput is one of the key factors to measure performance of an algorithm. Throughput will be given in the general form of completions per unit of time, the common throughput metric is instructions per cycle, In case of AES the throughput depends on size of block as well as time taken for encryption/decryption given by:

$$T = \frac{block\ size}{t}$$

Where,

T - Throughput

t - Time taken to encrypt/decrypt

2.4.3. Speedup

In the field of computer architecture, speedup is a metric for relative performance improvement when executing a task. The notion of speedup was established by Amdahl's law, which was particularly focused in the context of parallel processing. However, speedup can be used more generally to show the effect of any performance enhancement.

Speedup can be defined for two different types of values: throughput and latency. Throughput metric is instructions per cycle whereas the reciprocal of this is cycles per instruction or CPI; this is a latency quantity because it is the length of time between successive completions or occurrences.

Speedup is given by the following relation:

$$S = \frac{T_{old}}{T_{new}}$$

Where,

S is the resultant speedup.

T_{old} is the old execution time, i.e., without the improvement.

T_{new} is the new execution time, i.e., with the improvement.

Chapter 3

Literature Review

At present, there are many research achievements in the field of block cipher. Especially, the Advanced Encryption Standard AES algorithm should be considered the excellent representative of all the researches. When the data encryption standard was replaced by the advanced encryption standard, the whole world shifted their concern on the AES algorithm. Some research showed that the AES algorithm can be implemented with increased speed by shifting, XOR and looking up tables, etc. The analysis of some research work on AES algorithm based on increasing its speed and level of security by altering the parameters that have been described below:

Table 3: Research Analysis [6], [7], [8]

Author	Name	Year	Technique	Results
Deguang Le, Jinyi Chang, Xingdou Gou, Ankang Zhang, Conglan Lu	Parallel AES Algorithm for Fast Data Encryption on GPU	2010	Parallel encryption to design a fast data encryption system based on GPU.	Speedup= GPU_Time /CPU_Time (For plaintext sizes: 10KB Speedup=2 1MB Speedup=4 200MB Speedup=7)
Vishal Pachori, Gunjan Ansari, Neha Chaudhary	Improved Performance of Advance Encryption Standard using Parallel Computing	2012	Parallel Implementation of AES using Java Parallel Programming Framework	Speed up achieved for data parallelism and control parallelism is up to 2.16
Ritu Pahal, Vikas kumar	Efficient Implementation of AES	2013	The same conventional algorithm is implemented for 200 bit block as well as key size.	Encryption time decreased by 20% Throughput is : T=200/t (conventional being T=128/t)

3.1. Increasing the Block Size

Symmetric cryptography, such as in the Data Encryption Standard (DES), 3DES, and Advanced Encryption Standard (AES), uses an identical key for the sender and receiver, both to encrypt the message text and decrypt the cipher text. Symmetric cryptography is more suitable for the encryption of a large amount of data. The AES algorithm defined by the National Institute of Standards and Technology (NIST) of the United States has been widely accepted to replace DES as the new symmetric encryption algorithm. The AES algorithm is a symmetric block cipher that processes data blocks of 128 bits using a cipher key of length 128, 192, or 256 bits. Each data block consists of a 4 × 4 array of bytes called the state, on which the basic operations of the AES algorithm are performed.

The proposed algorithm differs from conventional AES [7] as it has 200 bits block size and key size both. Number of rounds is constant and equal to ten in this algorithm. The key expansion and substitution box generation are done in the same way as in conventional AES block cipher. AES has 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys and the same conventional 128 bit conventional AES algorithm is implemented for 200 bit using 5*5 Matrix. After the implementation, the proposed work is compared with 128 bit, 192 bits & 256 bits AES techniques on two points. These points are encryption and decryption time and throughput at both encryption and decryption sides.

At the start of encryption, 200 bit input is copied to the State array of 5*5 matrix. The data bytes are filled first in the column then in the rows. Then after the initial round key addition, ten rounds of encryption are performed. The first nine rounds are same, with small difference in the final round. Each of the first nine rounds consists of 4 transformations: SubBytes, ShiftRows, MixColumns and AddRoundKey. But in final round Mixcolumns transformation is not used.

- SubBytes Transformation - In this transformation, each of the byte in the state matrix is replaced with another byte as per the S-box. The S-box is generated by firstly calculating the respective reciprocal of that byte in GF (2^8) and then affine transform is applied.
- ShiftRows Transformation - In this transformation, the bytes in the first row of the State do not change. The second, third, fourth and fifth rows shift cyclically to the left by one byte, two bytes, three bytes and four bytes respectively.

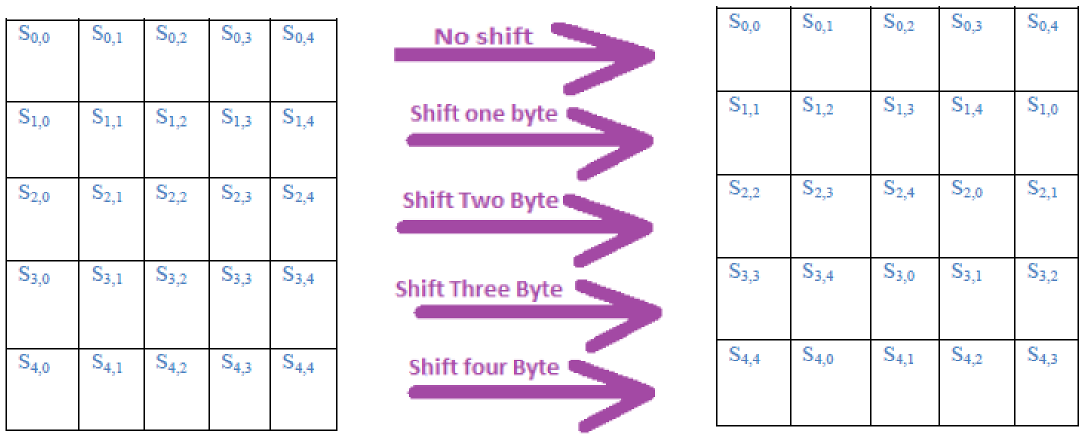

Figure 2: Shift Rows Transformation [7]

- MixColumns Transformation - It is the operation that mixes the bytes in each column by the multiplication of the state with a fixed polynomial matrix. It completely changes the scenario of the cipher even if the all bytes look very similar. The Inverse Polynomial Matrix does exist in order to reverse the mix column transformation.
- AddRoundKey Transformation - In AddRoundKey transformation, a round key is added to the State by bitwise Exclusive-OR (XOR) operation.

02	04	03	01	01
01	02	04	03	01
01	01	02	04	03
03	01	01	02	04
04	03	01	01	02

E0	7D	09	8A	4C
4C	E0	7D	09	8A
8A	4C	E0	7D	09
09	8A	4C	E0	7D
7D	09	8A	4C	E0

Figure 3: Polynomial Matrix and Its Inverse for mix column transformation [7]

The Decryption structure of proposed algorithm is obtained by inverting the encryption structure. Corresponding to the transformations in the encryption, InvSubBytes, InvShiftRows, InvMixColumns, and AddRoundKey are the transformations used in the decryption. The round keys are the same as those in encryption generated by Key Expansion, but are used in reverse order.

From the experimentation results it is deduced that for large block of data AES-200 encryption time per bit is reduced up to 20% and decryption time per bit is increased up to 25%. The throughput may be defined as number of bits that can be encrypted or decrypted during one unit of time. As it was mentioned earlier that all AES variant has equal block size of 128 bits and the proposed algorithm has block size of 200 bits. Thus, in form of equation the throughput may be defined as:

$$THRCA = \frac{128}{TENC}$$

$$THRPA = \frac{200}{TPENC}$$

Where, $THRCA$ is representation of throughput for conventional algorithms, $THRPA$ is representation of throughput for proposed algorithm, $TENC$ denotes the time taken to encrypt the 128 bit block message, ENC represents time taken to encrypt the 200 bit block message of conventional algorithm.

It is observed that the throughput at encryption end of AES-200 is 15% more than AES-128, 20% more than AES-192 and 30% more than AES-256. The decryption process of AES-200 is slower than conventional AES, the proposed algorithm is 50% slower from AES-128, 40% from AES-192, and 25% from AES-256.

3.2. Parallel Execution

To improve the performance of AES algorithm using parallel computing there are two major approaches Control Parallelism and Data Parallelism [8].

In Data Parallelism the data is divided into more than one part and send different part to different nodes for execution. Each node is executing the same procedure or function but on different data. This approach is very effective when there is large data to process. AES can be implemented in the following manner using DATA parallelism. Server sends Plaintext with the Key on node 1 and it will compute the cipher text by running the AES algorithm and finally sends the result back to the Server. Node 2 follows the same procedure. The number of nodes can be increased according to our requirement and number of processing units available.

In Control Parallelism the operation or function is divided instead of data. The different operation or function is assigned to different nodes and then finally the output is send to the server for final processing. Although it is less scalable then data parallelism but more speed up can be achieved by this approach. In control parallelism approach, the four main operations in AES algorithm are divided into two parts and combination of these operations is Operation 1 and Operation 2. Node 1 will execute only operation 1 and Node 2 will perform only operation 2. Nodes will communicate the result of each other when needed.

The performance of proposed architecture is measured in terms of execution time. The performance is measured on 256 bits of data and on two nodes or processing units. The execution time of converting 256 bits plain text into cipher text on Java Parallel Programming Framework using two nodes. The time taken by single core to encrypt 256 bits of data is 14, 15 and 13 seconds in different run. The time taken by the 1st run is more than the time taken in the subsequent run because in the first run the Hazelcast Framework is loaded which takes time to load. In the subsequent runs the time taken by the modified AES algorithm is almost same i.e. execution time gets stable. Speed up of the modified AES algorithm is shown below:

$$Speed\ Up = \frac{Time\ taken\ by\ serial\ algorithm}{Time\ taken\ by\ parallel\ algorithm}$$

Speed up for Data parallelism (1st run) = 15/10 = 1.5

Speed up for Data parallelism (2nd run) = 14/7 = 2.0

Speed up for Data parallelism (3rd run) = 13/6 = 2.16

Speed up for Data parallelism (4th run) = 13/7 = 1.85

Speed up for Control parallelism (1st run) = 15/11 = 1.36

Speed up for Control parallelism (2nd run) = 14/7 = 2.0

Speed up for Control parallelism (3rd run) = 13/6 = 2.16

Speed up for Control parallelism (4th run) = 13/6 = 2.16

In order to overcome the issue of low efficiency over the traditional CPU-based implementation of AES [6], researchers designed and implemented the parallel AES algorithm based on GPU. The implementation achieves up to 7x speedup over the implementation of AES on a comparable CPU. The implementation can be applied for the computer forensics which requires high speed of data encryption.

3.3. Threads

In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process. Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its instructions (executable code) and its context (the values of its variables at any given moment).

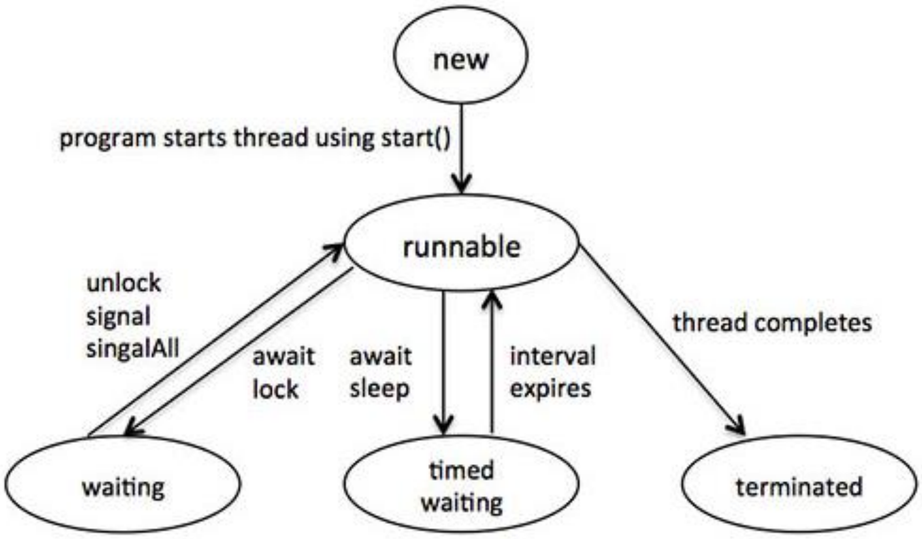

Figure 4: Life cycle of a thread [16]

The thread is in the "new" state, once it is constructed. In this state, it is merely an object in the heap, without any system resources allocated for execution. From the "new" state, the only thing you can do is to invoke the start() method, which puts the thread into the "runnable" state. Calling any method besides the start() will trigger an IllegalThreadStateException.

The start() method allocates the system resources necessary to execute the thread, schedules the thread to be run, and calls back the run() once it is scheduled. This put the thread into the "runnable" state. However, most computers have a single CPU and time-slice the CPU to support multithreading. Hence, in the "runnable" state, the thread may be running or waiting for its turn of the CPU time. A thread cannot be started twice, which triggers a runtime IllegalThreadStateException. The thread enters the "not-runnable" state when one of these events occurs: The sleep() method is called to suspend the thread for a specified amount of time to yield control to the other threads. You can also invoke the yield() to hint to the scheduler that the current thread is willing to yield its current use of a processor. The scheduler is, however, free to ignore this hint. The wait() method is called to wait for a specific condition to be satisfied. The thread is blocked and waiting for an I/O operation to be completed. For the "non-runnable" state, the thread becomes "runnable" again: If the thread was put to sleep, the specified sleep-time expired or the sleep was interrupted via a call to the interrupt() method. If the thread was put to wait via wait(), its notify() or notifyAll() method was invoked to inform the waiting thread that the specified condition had been fulfilled and the wait was over.If the thread was blocked for an I/O operation, the I/O operation has been completed.

A thread is in a "terminated" state, only when the run() method terminates naturally and exits. The method isAlive() can be used to test whether the thread is alive. The isAlive() returns false if the thread is "new" or "terminated". It returns true if the thread is "runnable" or "not-runnable". JDK 1.5 introduces a new getState() method. This method returns an (nested) enum of type Thread.State, which takes a constant of {NEW, BLOCKED, RUNNABLE, TERMINATED, WAITING}.

NEW: the thread has not yet started.

RUNNABLE:

WAITING:

BLOCKED: the thread is blocked waiting for a monitor lock.

TIMED_WAITING: the thread is waiting with a specified waiting time.

TERMINATED:

On a single processor, multithreading is generally implemented by time-division multiplexing (as in multitasking), and the central processing unit (CPU) switches between different software threads. This context switching generally happens frequently enough that the user perceives the threads or tasks as running at the same time. On a multiprocessor or multi-core system, threads can be executed in a true concurrent manner, with every processor or core executing a separate thread simultaneously. To implement multiprocessing, the operating system may use hardware threads that exist as a hardware-supported method for better utilization of a particular CPU, and are different from the software threads that are a pure software construct with no CPU-level representation.

Process schedulers of many modern operating systems directly support both time-sliced and multiprocessor threading. The operating system kernel allows programmers to manipulate threads by exposing required functionality through the system call interface. Some threading implementations are called kernel threads, whereas lightweight processes (LWP) are a specific type of kernel thread that share the same state and information.

Programs can have user-space threads when threading with timers, signals, or other methods to interrupt their own execution, performing a sort of ad hoc time-slicing.

We can think of a thread as basically a lightweight process. In order to understand this let us consider the two main characteristics of a process:

Unit of resource ownership

- A process is allocated:
- A virtual address space to hold the process image
- Control of some resources (files, I/O devices...)

Unit of dispatching

- A process is an execution path through one or more programs:
- Execution may be interleaved with other processes
- The process has an execution state and a dispatching priority

If we treat these two characteristics as being independent (as does modern OS theory):

The unit of resource ownership is usually referred to as a process or task. This Processes have:

- A virtual address space which holds the process image.
- Protected access to processors, other processes, files, and I/O resources.

The unit of dispatching is usually referred to a thread or a lightweight process. Thus a thread:

- Has an execution state (running, ready, etc.)
- Saves thread context when not running
- Has an execution stack and some per-thread static storage for local variables
- Has access to the memory address space and resources of its process

All threads of a process share this when one thread alters a (non-private) memory item, all other threads (of the process) sees that a file open with one thread, is available to others.

3.3.1. Benefits of Threads over Processes

A process runs independently and isolated of other processes. It cannot directly access shared data in other processes. The resources of the process, e.g. memory and CPU time, are allocated to it via the operating system.

A thread is a so called lightweight process. It has its own call stack, but can access shared data of other threads in the same process. Every thread has its own memory cache. If a thread reads shared data it stores this data in its own memory cache. A thread can re-read the shared data. A Java application runs by default in one process. Within a Java application you work with several threads to achieve parallel processing or asynchronous behavior.

Threads in the same process share the same address space. This allows concurrently running code to couple tightly and conveniently exchange data without the overhead or complexity of an inter process communication. When shared between threads, however, even simple data structures become prone to race conditions if they require more than one CPU instruction to update: two threads may end up attempting to update the data structure at the same time and find it unexpectedly changing underfoot. Bugs caused by race conditions can be very difficult to reproduce and isolate.

- Simpler Program Design: If you were to program the above ordering of reading and processing by hand in a single threaded application, you would have to keep track of both the read and processing state of each file. Instead you can start two threads that each just reads and processes a single file. Each of these threads will be blocked while waiting for the disk to read its file. While waiting, other threads can use the CPU to process the parts of the file they have already read. The result is, that the disk is kept busy at all times, reading from various files into memory. This results in a better utilization of both the disk and the CPU. It is also easier to program, since each thread only has to keep track of a single file.
- More responsive programs: Another common goal for turning a single threaded application into a multithreaded application is to achieve a more responsive application. Imagine a server application that listens on some port for incoming requests. When a request is received, it handles the request and then goes back to listening. If the request takes a long time to process, no new clients can send requests to the server for that duration. Only while the server is listening can requests be received.

An alternate design would be for the listening thread to pass the request to a worker thread, and return to listening immediately. The worker thread will process the request and send a reply to the client. This way the server thread will be back at listening sooner. Thus more clients can send requests to the server. The server has become more responsive.

The same is true for desktop applications. If you click a button that starts a long task, and the thread executing the task is the thread updating the windows, buttons etc., then the application will appear unresponsive while the task executes. Instead the task can be handed off to a worker thread. While the worker thread is busy with the task, the window thread is free to respond to other user requests. When the worker thread is done it signals the window thread. The window thread can then update the application windows with the result of the task. The program with the worker thread design will appear more responsive to the user.

If implemented correctly then threads have some advantages of (multi) processes, they take:

- Less time to create a new thread than a process, because the newly created thread uses the current process address space.
- Less time to terminate a thread than a process.
- Less time to switch between two threads within the same process, partly because the newly created thread uses the current process address space.
- Less communication overheads -- communicating between the threads of one process is simple because the threads share everything: address space, in particular. So, data produced by one thread is immediately available to all the other threads.

3.3.2 Multithreading vs Single Threading

Multithreading is mainly found in multitasking operating systems. Multithreading is a widespread programming and execution model that allows multiple threads to exist within the context of a single process. These threads share the process's resources, but are able to execute independently. The threaded programming model provides developers with a useful abstraction of concurrent execution. Multithreading can also be applied to a single process to enable parallel execution on a multiprocessing system.

Multithreaded applications have the following advantages:

- Responsiveness: Multi-threading can allow an application to remain responsive to input. In a single-threaded program, if the main execution thread blocks on a long-running task, the entire application can appear to freeze. By moving such long-running tasks to a worker thread that runs concurrently with the main execution thread, it is possible for the application to remain responsive to user input while executing tasks in the background. On the other hand, in most cases multithreading is not the only way to keep a program responsive, with non-blocking I/O and/or Unix signals being available for gaining similar results.
- Faster execution: This advantage of a multithreaded program allows it to operate faster on computer systems that have multiple or multi-core CPUs, or across a cluster of machines, because the threads of the program naturally lend themselves to truly concurrent execution.

- Lower resource consumption: Using threads, an application can serve multiple clients concurrently using fewer resources than it would need when using multiple process copies of itself. For example, the Apache HTTP server, which uses a pool of listener threads for listening to incoming requests and a pool of server threads for processing those requests.
- Better system utilization: As an example, a file-system using multiple threads can achieve higher throughput and lower latency since data in a faster medium (such as cache memory) can be retrieved by one thread while another thread retrieves data from a slower medium (such as external storage) without either thread waitng for the other to complete.
- Simplified sharing and communication: Unlike processes, which require a message passing or shared memory mechanism to perform inter-process communication, threads can communicate through data, code and files that they already share.
- Parallelization: Applications looking to utilize multi-core and multi-CPU systems can use multi-threading to split data and tasks into parallel sub-tasks and let the underlying architecture manage how the threads run, either concurrently on a single core or in parallel on multiple cores. GPU computing environments like CUDA and OpenCL use the multi-threading model where dozens to hundreds of threads run in parallel on a large number of cores.

Some of the Single Threading Benefits are:

- Programming and debugging - These activities are easier compared to multithreaded applications due to the reduced complexity.
- Less Overhead - Threads add overhead to an application.

When developing multi-threaded applications, the following must be considered:

- Deadlocks occur when two threads hold a monitor that the other one requires. In essence each task is blocking the other and both tasks are waiting for the other monitor to be released. This forces an application to hang or deadlock.
- Resource allocation is used to prevent deadlocks because the system determines if approving the resource request will render the system in an unsafe state. An unsafe state could result in a deadlock. The system only approves requests that will lead to safe states.

- Thread Synchronization is used when multiple threads use the same instance of an object. The threads accessing the object can then be locked and then synchronized so that each task can interact with the static object on at a time.

Multithreading has the following drawbacks:

- Synchronization: Since threads share the same address space, the programmer must be careful to avoid race conditions and other non-intuitive behaviors. In order for data to be correctly manipulated, threads will often need to rendezvous in time in order to process the data in the correct order. Threads may also require mutually exclusive operations (often implemented using semaphores) in order to prevent common data from being simultaneously modified or read while in the process of being modified. Careless use of such primitives can lead to deadlocks.
- Thread crashes a process: An illegal operation performed by a thread crashes the entire process; therefore, one misbehaving thread can disrupt the processing of all the other threads in the application.
- Multiple threads can interfere with each other when sharing hardware resources such as caches or translation lookaside buffers (TLBs).
- Execution times of a single thread are not improved but can be degraded, even when only one thread is executing. This is due to slower frequencies and/or additional pipeline stages that are necessary to accommodate thread-switching hardware.
- Hardware support for multithreading is more visible to software, thus requiring more changes to both application programs and operating systems than multiprocessing.

A process or program has its own address space and control blocks. It is called heavyweight because it consumes a lot of system resources. Within a process or program, we can run multiple threads concurrently to improve the performance.

Threads, unlike heavyweight process, are lightweight and run inside a single process - they share the same address space, the resources allocated and the environment of that process. It is lightweight because it runs within the context of a heavyweight process and takes advantage of the resources allocated for that program and the program’s environment. A thread must carve out its own resources within the running process. For example, a thread has its own stack, registers and program counter. The code running within the thread works only within that context, hence, a thread (of a sequential flow of operations) is also called an execution context.

Multithreading within a program improves the performance of the program by optimizing the usage of system resources. For example, while one thread is blocked (e.g., waiting for completion of an I/O operation), another thread can use the CPU time to perform computations, resulted in better performance and overall throughput.

Multithreading is also necessary to provide better interactivity with the users. For example, in a word processor, while one thread is printing or saving the file, another thread can be used to continue typing. In GUI applications, multithreading is essential in providing a responsive user interface.

A typical Java program runs in a single process, and is not interested in multiple processes. However, within the process, it often uses multiple threads to run multiple tasks concurrently. A standalone Java application starts with a single thread (called main thread) associated with the main() method. This main thread can then start new user threads.

3.3.3 Synchronization

Thread synchronization is defined as a mechanism which ensures that two or more concurrent processes or threads do not simultaneously execute some particular program segment known as mutual exclusion. When one thread starts executing the critical section (serialized segment of the program) the other thread should wait until the first thread finishes. If proper synchronization techniques are not applied, it may cause a race condition where, the values of variables may be unpredictable and vary depending on the timings of context switches of the processes or threads.

A way of making sure that if one process is using a shared modifiable data, the other processes will be excluded from doing the same thing. Formally, while one process executes the shared variable, all other processes desiring to do so at the same time moment should be kept waiting; when that process has finished executing the shared variable, one of the processes waiting; while that process has finished executing the shared variable, one of the processes waiting to do so should be allowed to proceed. In this fashion, each process executing the shared data (variables) excludes all others from doing so simultaneously. This is called Mutual Exclusion. Mutual Exclusion needs to be enforced only when processes access shared modifiable data - when processes are performing operations that do not conflict with one another they should be allowed to proceed concurrently.

Mutual Exclusion Conditions:

- If we could arrange matters such that no two processes were ever in their critical sections simultaneously, we could avoid race conditions. We need four conditions to hold to have a good solution for the critical section problem (mutual exclusion).
- No two processes may at the same moment inside their critical sections.
- No assumptions are made about relative speeds of processes or number of CPUs.
- No process should outside its critical section should block other processes.
- No process should wait arbitrary long to enter its critical section.

Other than mutual exclusion, synchronization also deals with the following:

- Deadlock: This occurs when many processes are waiting for a shared resource (critical section) which is being held by some other process. In this case the processes just keep waiting and execute no further.
- Starvation: A process is waiting to enter the critical section but other processes keep on executing the critical section and the first process just keeps on waiting.
- Priority inversion: When a high priority process is in the critical section, it may be interrupted by a medium priority process. This is the violation of rules BUT this may happen and may lead to some serious consequences when dealing with real-time problems.
- Busy waiting: It occurs when a process is waiting for its turn but simultaneously it is continuously checking that now its turn to process or not. This checking is basically robbing the processing time of other processes.

Processes access to critical section is controlled by using synchronization techniques.

In Java, there are two common synchronization strategies to prevent thread interference and memory consistency errors:

- Synchronized Method: It includes the synchronized keyword in the declaration of the method. So when any thread invokes this synchronized method, that method acquires the intrinsic lock by its own (automatically) for that method's object and it releases the lock when the method returns, even if the return was caused by some uncaught exception.
- Synchronized Statement: Here we declare a block of code to be synchronized. Unlike synchronized methods, synchronized statements need to specify the objects that provide the intrinsic lock. To improve the concurrency with fine-grained synchronization, synchronized statements are very useful because they prevent unnecessary blocking.

Chapter 4

Design and Implementation

AES algorithm is the current standard for symmetric key encryption, this section gives a detailed explanation about the various permutation and substitution steps followed in order to perform encryption and decryption.

4.1. Detailed Description

The following is the brief overview of various terminologies used in implementation of the AES algorithm:

4.1.1. Terminology

State: Defines the current condition (state) of the block. That is the block of bytes that are currently being worked on. The state starts off being equal to the block, however it changes as each round of the algorithms executes. Plainly said this is the block in progress.

Hex to Decimal table:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

For example using the above table HEX D4 = DEC 212

Figure 5: HEX Matrix [9]

Block: AES is a block cipher. This means that the number of bytes that it encrypts is fixed. AES can currently encrypt blocks of w 16 bytes at a time; no other block sizes are presently a part of the AES standard. If the bytes being encrypted are larger than the specified block then AES is executed concurrently. This also means that AES has to encrypt a minimum of 16 bytes. If the plain text is smaller than 16 bytes then it must be padded. Simply said the block is a reference to the bytes that are processed by the algorithm.

HEX: Defines a notation of numbers in base 16. This simply means that; the highest number that can be represented in a single digit is 15, rather than the usual 9 in the decimal (base 10) system.

XOR: Refers to the bitwise operator Exclusive Or. XOR operates on the individual bits in a byte in the following way:

0 XOR 0 = 0

1 XOR 0 = 1

1 XOR 1 = 0

0 XOR 1 = 1

Most programming languages have the XOR operator built in. Another interesting property of the XOR operator is that it is reversible.

So Hex 2B XOR FF = D4. AES is an iterated symmetric block cipher, which means that:

- AES works by repeating the same defined steps multiple times.
- AES is a secret key encryption algorithm.
- AES operates on a fixed number of bytes

AES as well as most encryption algorithms is reversible. This means that almost the same steps are performed to complete both encryption and decryption in reverse order. The AES algorithm operates on bytes, which makes it simpler to implement and explain. This key is expanded into individual sub keys, a sub keys for each operation round. This process is called Key Expansion, which is described at the end of this document. As mentioned before AES is an iterated block cipher. All that means is that the same operations are performed many times on a fixed number of bytes. These operations can easily be broken down to the following functions:

- ADD ROUND KEY
- SUB BYTE
- SHIFT ROW
- MIX COLUMN

An iteration of the above steps is called a round. The amount of rounds of the algorithm depends on the key size. The only exception being that in the last round the Mix Column step is not performed to make the algorithm reversible during decryption.

Table 4: Number of rounds for various key sizes [3]

Key Size (Bytes)	**Block Size (Bytes)**	**Rounds**
16	16	10
24	16	12
32	16	14

Encryption

The following tables illustrates the number of rounds required for encryption depending on different key size length:

Table 5: AES Encryption cipher using 16-bit key [3]

Round	**Function**
-	Add Round Key(State)
1	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
2	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
3	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
4	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
5	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
6	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
7	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
8	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
9	Add Round Key(Shift Row(Byte Sub(State)))

Table 6: AES Encryption cipher using 24-bit key [3]

Round	Function
-	Add Round Key(State)
1	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
2	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
3	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
4	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
5	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
6	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
7	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
8	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
9	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
10	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
11	Add Round Key(Shift Row(Byte Sub(State)))

Table 7: AES Encryption cipher using 32-bit key [3]

Round	Function
-	Add Round Key(State)
1	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
2	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
3	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
4	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
5	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
6	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
7	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
8	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
9	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
10	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
11	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
12	Add Round Key(Mix Column(Shift Row(Byte Sub(State))))
13	Add Round Key(Shift Row(Byte Sub(State)))

Decryption

The following tables illustrates the number of rounds required for encryption depending on different key size length:

Table 8: AES Decryption cipher using 16-bit key [3]

Round	Function
-	Add Round Key(State)
1	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
2	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
3	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
4	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
5	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
6	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
7	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
8	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
9	Add Round Key(Byte Sub(Shift Row(State)))

Table 9: AES Decryption cipher using 24-bit key [3]

Round	Function
-	Add Round Key(State)
1	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
2	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
3	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
4	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
5	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
6	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
7	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
8	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
9	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
10	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
11	Add Round Key(Byte Sub(Shift Row(State)))

Table 10: AES Decryption cipher using 32-bit key [3]

Round	Function
-	Add Round Key(State)
1	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
2	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
3	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
4	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
5	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
6	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
7	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
8	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
9	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
10	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
11	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
12	Mix Column(Add Round Key(Byte Sub(Shift Row(State))))
13	Add Round Key(Byte Sub(Shift Row(State)))

4.2. AES Cipher Functions

Given below is the detailed description of all the 4 functions and the corresponding inverse functions that are used in various rounds of encryption as well as decryption process:

4.2.1. Add Round Key

Each of the 16 bytes of the state is XORed against each of the 16 bytes of a portion of the expanded key for the current round.

The first time Add Round Key gets executed

State	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR
Exp Key	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

The second time Add Round Key is executed

State	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR
Exp Key	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32

And so on for each round of execution.

Figure 6: Working of Add Round Key [1]

The Expanded Key bytes are never reused. So once the first 16 bytes are XORed against the first16 bytes of the expanded key then the expanded key bytes 1-16 are never used again. The next time the AddRound Key function is called bytes 17-32 are XORed against the state.

4.2.2. Byte Sub

During encryption each value of the state is replaced with the corresponding SBOX value.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
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

Figure 7: SBOX [1]

For example HEX 19 would get replaced with HEX D4

Whereas during decryption each value in the state is replaced with the corresponding inverse of the SBOX.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
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 8: Inverse SBOX [1]

For example HEX D4 would get replaced with HEX 19

4.2.3. Shift Row

Arranges the state in a matrix and then performs a circular shift for each row. This is not a bit wise shift. The circular shift just moves each byte one space over. A byte that was in the second position may end up in the third position after the shift. The circular part of it specifies that the byte in the last position shifted one space will end up in the first position in the same row.

In Detail:

- The state is arranged in a 4x4 matrix (square)
- The confusing part is that the matrix is formed vertically but shifted horizontally. So the first 4 bytes of the state will form the first bytes in each row. So bytes 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Will form a matrix:

1 5 9 13

2 6 10 14

3 7 11 15

4 8 12 16

Each row is then moved over (shifted) 1, 2 or 3 spaces over to the right, depending on the row of the state. First row is never shifted

Row1 0

Row2 1

Row3 2

Row4 3

The following is the illustration of how the individual bytes are first arranged in the table and then moved over (shifted).

Blocks 16 bytes long:

From	To
1 5 9 13	1 5 9 13
2 6 10 14	6 10 14 2
3 7 11 15	11 15 3 7
4 8 12 16	16 4 8 12

During decryption the same process is reversed and all rows are shifted to the left:

From	To
1 5 9 13	1 5 9 13
2 6 10 14	14 2 6 10
3 7 11 15	11 15 3 7
4 8 12 16	8 12 16 4

4.2.4. Mix Column

This is perhaps the hardest step to both understand and explain. There are two parts to this step. The first will explain which parts of the state are multiplied against which parts of the matrix.

Matrix Multiplication:

The state is arranged into a 4 row table (as described in the Shift Row function).

The multiplication is performed one column at a time (4 bytes). Each value in the column is eventually multiplied against every value of the matrix (16 total multiplications). The results of these multiplications are XORed together to produce only 4 result bytes for the next state. Therefore 4 bytes input, 16 multiplications 12 XORs and 4 bytes output. The multiplication is performed one matrix row at a time against each value of a state column.

Multiplication Matrix

2 3 1 1

1 2 3 1

1 1 2 3

3 1 1 2

16 byte State

b1 b5 b9 b13

b2 b6 b10 b14

b3 b7 b11 b15

b4 b8 b12 b16

The first result byte is calculated by multiplying 4 values of the state column against 4 values of the first row of the matrix. The result of each multiplication is then XORed to produce 1 byte:

b1 = (b1 * 2) XOR (b2*3) XOR (b3*1) XOR (b4*1)

The second result byte is calculated by multiplying the same 4 values of the state column against 4 values of the second row of the matrix. The result of each multiplication is then XORed to produce 1 byte:

b2 = (b1 * 1) XOR (b2*2) XOR (b3*3) XOR (b4*1)

The third result byte is calculated by multiplying the same 4 values of the state column against 4 values of the third row of the matrix. The result of each multiplication is then XORed to produce 1 byte:

b3 = (b1 * 1) XOR (b2*1) XOR (b3*2) XOR (b4*3)

The fourth result byte is calculated by multiplying the same 4 values of the state column against 4 values of the fourth row of the matrix. The result of each multiplication is then XORed to produce 1 byte:

b4 = (b1 * 3) XOR (b2*1) XOR (b3*1) XOR (b4*2)

This procedure is repeated again with the next column of the state, until there are no more state columns.

Putting it all together:

The first column will include state bytes 1-4 and will be multiplied against the matrix in the following manner:

b1 = (b1 * 2) XOR (b2*3) XOR (b3*1) XOR (b4*1)

b2 = (b1 * 1) XOR (b2*2) XOR (b3*3) XOR (b4*1)

b3 = (b1 * 1) XOR (b2*1) XOR (b3*2) XOR (b4*3)

b4 = (b1 * 3) XOR (b2*1) XOR (b3*1) XOR (b4*2)

(b1= specifies the first byte of the state)

The second column will be multiplied against the second row of the matrix in the following manner.

b5 = (b5 * 2) XOR (b6*3) XOR (b7*1) XOR (b8*1)

b6 = (b5 * 1) XOR (b6*2) XOR (b7*3) XOR (b8*1)

b7 = (b5 * 1) XOR (b6*1) XOR (b7*2) XOR (b8*3)

b8 = (b5 * 3) XOR (b6*1) XOR (b7*1) XOR (b8*2)

And so on until all columns of the state are exhausted.

4.2.5. Mix Column Inverse

During decryption the Mix Column the multiplication matrix is changed to:

0E 0B 0D 09

09 0E 0B 0D

0D 09 0E 0B

0B 0D 09 0E

Apart from the change to the matrix table the function performs the same steps as during encryption.

Mix Column Example

The following examples are denoted in HEX.

- Mix Column Example during Encryption

Input = D4 BF 5D 30

Output(0) = (D4 * 2) XOR (BF*3) XOR (5D*1) XOR (30*1)

= E(L(D4) + L(02)) XOR E(L(BF) + L(03)) XOR 5D XOR 30

= E(41 + 19) XOR E(9D + 01) XOR 5D XOR 30

= E(5A) XOR E(9E) XOR 5D XOR 3010

= B3 XOR DA XOR 5D XOR 30

= 04

Output(1) = (D4 * 1) XOR (BF*2) XOR (5D*3) XOR (30*1)

= D4 XOR E(L(BF)+L(02)) XOR E(L(5D)+L(03)) XOR 30

= D4 XOR E(9D+19) XOR E(88+01) XOR 30

= D4 XOR E(B6) XOR E(89) XOR 30

= D4 XOR 65 XOR E7 XOR 30

= 66

Output(2) = (D4 * 1) XOR (BF*1) XOR (5D*2) XOR (30*3)

= D4 XOR BF XOR E(L(5D)+L(02)) XOR E(L(30)+L(03))

= D4 XOR BF XOR E(88+19) XOR E(65+01)

= D4 XOR BF XOR E(A1) XOR E(66)

= D4 XOR BF XOR BA XOR 50

= 81

Output(3) = (D4 * 3) XOR (BF*1) XOR (5D*1) XOR (30*2)

= E(L(D4)+L(3)) XOR BF XOR 5D XOR E(L(30)+L(02))

= E(41+01) XOR BF XOR 5D XOR E(65+19)

= E(42) XOR BF XOR 5D XOR E(7E)

= 67 XOR BF XOR 5D XOR 60

= E5

- Mix Column during Decryption

Input 04 66 81 E5

Output(0) = (04 * 0E) XOR (66*0B) XOR (81*0D) XOR (E5*09)

=E(L(04)+L(0E)) XOR E(L(66)+L(0B)) XOR E(L(81)+L(0D)) XOR E(L(E5)+L(09))

= E(32+DF) XOR E(1E+68) XOR E(58+EE) XOR E(20+C7)

= E(111-FF) XOR E(86) XOR E(146-FF) XOR E(E7)

= E(12) XOR E(86) XOR E(47) XOR E(E7)

= 38 XOR B7 XOR D7 XOR 8C

= D4

Output(1) = (04 * 09) XOR (66*0E) XOR (81*0B) XOR (E5*0D)

= E(L(04)+L(09)) XOR E(L(66)+L(0E)) XOR E(L(81)+L(0B)) XOR E(L(E5)+L(0D))

= E(32+C7) XOR E(1E+DF) XOR E(58+68) XOR E(20+ EE)

= E(F9) XOR E(FD) XOR E(C0) XOR E(10E-FF)

= E(F9) XOR E(FD) XOR E(C0) XOR E(0F)

= 24 XOR 52 XOR FC XOR 35

= BF

Output(2) = (04 * 0D) XOR (66*09) XOR (81*0E) XOR (E5*0B)

=E(L(04)+L(0D)) XOR E(L(66)+L(09) XOR E(L(81)+L(0E)) XOR E(L(E5)+(0B))

= E(32+EE) XOR E(1E+C7) XOR E(58+DF) XOR E(20+68)

= E(120-FF) XOR E(E5) XOR E(137-FF) XOR E(88)

= E(21) XOR E(E5) XOR E(38) XOR E(88)

= 34 XOR 7B XOR 4F XOR 5D

= 5D

Output(3) = (04 * 0B) XOR (66*0D) XOR (81*09) XOR (E5*0E)

= E(L(04)+L(0B)) XOR E(L(66)+L(0D)) XOR E(L(81)+L(09)) XOR E(L(E5)+L(0E))

= E(32+68) XOR E(1E+EE) XOR E(58+C7) XOR E(20+DF)

= E(9A) XOR E(10C-FF) XOR E(11F-FF) XOR E(FF)

= E(9A) XOR E(0D) XOR E(20) XOR E(FF)

= 2C XOR F8 XOR E5 XOR 01

= 30

4.2.6. Key Expansion

Prior to encryption or decryption the key must be expanded. The expanded key is used in the Add Round Key function defined above. Each time the Add Round Key function is called a different part of the expanded key is XORed against the state. In order for this to work the Expanded Key must be large enough so that it can provide key material for every time the AddRoundKey function is executed. The Add Round Key function gets called for each round as well as one extra time at the beginning of the algorithm.

Therefore the size of the expanded key will always be equal to:

16 * (number of rounds + 1).

The 16 in the above function is actually the size of the block in bytes. This provides key material for every byte in the block during every round +1

Since the key size is much smaller than the size of the sub keys, the key is actually stretched out to provide enough key space for the algorithm. The key expansion routine executes a maximum of 4 consecutive functions. These functions are:

ROT WORD

SUB WORD

RCON

EK

K

An iteration of the above steps is called a round. The amount of rounds of the key expansion algorithm depends on the key size.

Table 11: Key Expansion [3]

Key Size (bytes)	Block Size (bytes)	Expansion Algorithm Rounds	Expanded Bytes / Round	Rounds Key Copy	Rounds Key Expansion	Expanded Key (bytes)
16	16	44	4	4	40	176
24	16	52	4	6	46	208
32	16	60	4	8	52	240

The first bytes of the expanded key are always equal to the key. If the key is 16 bytes long the first 16 bytes of the expanded key will be the same as the original key. If the key size is 32 bytes then the first 32 bytes of the expanded key will be the same as the original key.

Each round adds 4 bytes to the Expanded Key. With the exception of the first rounds each round also takes the previous rounds 4 bytes as input operates and returns 4 bytes. One more important note is that not all of the 4 functions are always called in each round. The algorithm only calls all 4 of the functions every:

4 Rounds for a 16 byte Key

6 Rounds for a 24 byte Key

8 Rounds for a 32 byte Key

The rest of the rounds only a K function result is XORed with the result of the EK function. There is an exception of this rule where if the key is 32 bytes long an additional call to the Sub Word function is called every 8 rounds starting on the 13th round.

Key Expansion Functions

The following are the various functions used in expanding the given key:

- **Rot Word (4 bytes)**

This does a circular shift on 4 bytes similar to the Shift Row Function.

1,2,3,4 to 2,3,4,1

- **Sub Word (4 bytes)**

This step applies the S-box value substitution as described in Bytes Sub function to each of the 4 bytes in the argument.

Rcon((Round/(KeySize/4))-1)

This function returns a 4 byte value based on the following table

Rcon(0) = 01000000

Rcon(1) = 02000000

Rcon(2) = 04000000

Rcon(3) = 08000000

Rcon(4) = 10000000

Rcon(5) = 20000000

Rcon(6) = 40000000

Rcon(7) = 80000000

Rcon(8) = 1B000000

Rcon(9) = 36000000

Rcon(10) = 6C000000

Rcon(11) = D8000000

Rcon(12) = AB000000

Rcon(13) = 4D000000

Rcon(14) = 9A000000

For example for a 16 byte key Rcon is first called in the 4th round

(4/(16/4))-1=0

In this case Rcon will return 01000000

For a 24 byte key Rcon is first called in the 6th round

(6/(24/4))-1=0

In this case Rcon will also return 01000000

- **EK(Offset)**

EK function returns 4 bytes of the Expanded Key after the specified offset. For example if offset is 0 then EK will return bytes 0,1,2,3 of the Expanded Key

- **K(Offset)**

K function returns 4 bytes of the Key after the specified offset. For example if offset is 0 then K will return bytes 0,1,2,3 of the Expanded Key

Since the expansion algorithm changes depending on the length of the key, it is extremely difficult to explain in writing. This is why the explanation of the Key Expansion Algorithm is provided in a table format.

- 16 byte Key Expansion:

Each round (except rounds 0, 1, 2 and 3) will take the result of the previous round and produce a 4 byte result for the current round. Notice the first 4 rounds simply copy the total of 16 bytes of the key.

Table 12: 16-byte key expansion [9]

Round	Expanded Key Bytes	Function
0	0 1 2 3	**K**(0)
1	4 5 6 7	**K**(4)
2	8 9 10 11	**K**(8)
3	12 13 14 15	**K**(12)
4	16 17 18 19	**Sub Word**(**Rot Word**(**EK**((4-1)*4))) XOR **Rcon**((4/4)-1) XOR **EK**((4-4)*4)
5	20 21 22 23	**EK**((5-1)*4)XOR **EK**((5-4)*4)
6	24 25 26 27	**EK**((6-1)*4)XOR **EK**((6-4)*4)
7	28 29 30 31	**EK**((7-1)*4)XOR **EK**((7-4)*4)
8	32 33 34 35	**Sub Word**(**Rot Word**(**EK**((8-4)*4))) XOR **Rcon**((8/4)-1) XOR **EK**((8-4)*4)
9	36 37 38 39	**EK**((8-1)*4)XOR **EK**((9-4)*4)
10	40 41 42 43	**EK**((10-1)*4)XOR **EK**((10-4)*4)
11	44 45 46 47	**EK**((11-1)*4)XOR **EK**((11-4)*4)
12	48 49 50 51	**Sub Word**(**Rot Word**(**EK**((12-4)*4))) XOR **Rcon**((12/4)-1) XOR **EK**((12-4)*4)
13	52 53 54 55	**EK**((13-1)*4)XOR **EK**((13-4)*4)
14	56 57 58 59	**EK**((14-1)*4)XOR **EK**((14-4)*4)
15	60 61 62 63	**EK**((15-1)*4)XOR **EK**((15-4)*4)
16	64 65 66 67	**Sub Word**(**Rot Word**(**EK**((16-4)*4))) XOR **Rcon**((16/4)-1) XOR **EK**((16-4)*4)
17	68 69 70 71	**EK**((17-1)*4)XOR **EK**((17-4)*4)
18	72 73 74 75	**EK**((18-1)*4)XOR **EK**((18-4)*4)
19	76 77 78 79	**EK**((19-1)*4)XOR **EK**((19-4)*4)
20	80 81 82 83	**Sub Word**(**Rot Word**(**EK**((20-4)*4))) XOR **Rcon**((20/4)-1) XOR **EK**((20-4)*4)
21	84 85 86 87	**EK**((21-1)*4)XOR **EK**((21-4)*4)
22	88 89 90 91	**EK**((22-1)*4)XOR **EK**((22-4)*4)
23	92 93 94 95	**EK**((23-1)*4)XOR **EK**((23-4)*4)
24	96 97 98 99	**Sub Word**(**Rot Word**(**EK**((24-4)*4))) XOR **Rcon**((24/4)-1) XOR **EK**((24-4)*4)
25	100 101 102 103	**EK**((25-1)*4)XOR **EK**((25-4)*4)
26	104 105 106 107	**EK**((26-1)*4)XOR **EK**((26-4)*4)
27	108 109 110 111	**EK**((27-1)*4)XOR **EK**((27-4)*4)
28	112 113 114 115	**Sub Word**(**Rot Word**(**EK**((28-4)*4))) XOR **Rcon**((28/4)-1) XOR **EK**((28-4)*4)
29	116 117 118 119	**EK**((29-1)*4)XOR **EK**((29-4)*4)
30	120 121 122 123	**EK**((30-1)*4)XOR **EK**((30-4)*4)
31	124 125 126 127	**EK**((31-1)*4)XOR **EK**((31-4)*4)
32	128 129 130 131	**Sub Word**(**Rot Word**(**EK**((32-4)*4))) XOR **Rcon**((32/4)-1) XOR **EK**((32-4)*4)
33	132 133 134 135	**EK**((33-1)*4)XOR **EK**((33-4)*4)
34	136 137 138 139	**EK**((34-1)*4)XOR **EK**((34-4)*4)
35	140 141 142 143	**EK**((35-1)*4)XOR **EK**((35-4)*4)
36	144 145 146 147	**Sub Word**(**Rot Word**(**EK**((36-4)*4))) XOR **Rcon**((36/4)-1) XOR **EK**((36-4)*4)
37	148 149 150 151	**EK**((37-1)*4)XOR **EK**((37-4)*4)
38	152 153 154 155	**EK**((38-1)*4)XOR **EK**((38-4)*4)
39	156 157 158 159	**EK**((39-1)*4)XOR **EK**((39-4)*4)
40	160 161 162 163	**Sub Word**(**Rot Word**(EK((40-4)*4))) XOR **Rcon**((40/4)-1) XOR **EK**((40-4)*4)
41	164 165 166 167	**EK**((41-1)*4)XOR **EK**((41-4)*4)
42	168 169 170 171	**EK**((42-1)*4)XOR **EK**((42-4)*4)
43	172 173 174 175	**EK**((43-1)*4)XOR **EK**((43-4)*4)

- 24 byte Key Expansion

Each round (except rounds 0, 1, 2, 3, 4 and 5) will take the result of the previous round and produce a 4 byte result for the current round. Notice the first 6 rounds simply copy the total of 24 bytes of the key.

Table 13: 24-byte key expansion [9]

Round	Expanded Key Bytes	Function
0	0 1 2 3	**K**(0)
1	4 5 6 7	**K**(4)
2	8 9 10 11	**K**(8)
3	12 13 14 15	**K**(12)
4	16 17 18 19	**K**(16)
5	20 21 22 23	**K**(20)
6	24 25 26 27	**Sub Word**(**Rot Word**(**EK**((6-1)*4))) XOR **Rcon**((6/6)-1) XOR **EK**((6-6)*4)
7	28 29 30 31	**EK**((7-1)*4)XOR **EK**((7-6)*4)
8	32 33 34 35	**EK**((8-1)*4)XOR **EK**((8-6)*4)
9	36 37 38 39	**EK**((9-1)*4)XOR **EK**((9-6)*4)
10	40 41 42 43	**EK**((10-1)*4)XOR **EK**((10-6)*4)
11	44 45 46 47	**EK**((11-1)*4)XOR **EK**((11-6)*4)
12	48 49 50 51	**Sub Word**(**Rot Word**(**EK**((12-1)*4))) XOR **Rcon**((12/6)-1) XOR **EK**((12-6)*4)
13	52 53 54 55	**EK**((13-1)*4)XOR **EK**((13-6)*4)
14	56 57 58 59	**EK**((14-1)*4)XOR **EK**((14-6)*4)
15	60 61 62 63	**EK**((15-1)*4)XOR **EK**((15-6)*4)
16	64 65 66 67	**EK**((16-1)*4)XOR **EK**((16-6)*4)
17	68 69 70 71	**EK**((17-1)*4)XOR **EK**((17-6)*4)
18	72 73 74 75	**Sub Word**(**Rot Word**(**EK**((18-1)*4))) XOR **Rcon**((18/6)-1) XOR **EK**((18-6)*4)
19	76 77 78 79	**EK**((19-1)*4)XOR **EK**((19-6)*4)
20	80 81 82 83	**EK**((20-1)*4)XOR **EK**((20-6)*4)
21	84 85 86 87	**EK**((21-1)*4)XOR **EK**((21-6)*4)
22	88 89 90 91	**EK**((22-1)*4)XOR **EK**((22-6)*4)
23	92 93 94 95	**EK**((23-1)*4)XOR **EK**((23-6)*4)
24	96 97 98 99	**Sub Word**(**Rot Word**(**EK**((24-1)*4))) XOR **Rcon**((24/6)-1) XOR **EK**((24-6)*4)
25	100 101 102 103	**EK**((25-1)*4)XOR **EK**((25-6)*4)
26	104 105 106 107	**EK**((26-1)*4)XOR **EK**((26-6)*4)
27	108 109 110 111	**EK**((27-1)*4)XOR **EK**((27-6)*4)
28	112 113 114 115	**EK**((28-1)*4)XOR **EK**((28-6)*4)
29	116 117 118 119	**EK**((29-1)*4)XOR **EK**((29-6)*4)
30	120 121 122 123	**Sub Word**(**Rot Word**(**EK**((30-1)*4))) XOR **Rcon**((30/6)-1) XOR **EK**((30-6)*4)
31	124 125 126 127	**EK**((31-1)*4)XOR **EK**((31-6)*4)
32	128 129 130 131	**EK**((32-1)*4)XOR **EK**((32-6)*4)
33	132 133 134 135	**EK**((33-1)*4)XOR **EK**((33-6)*4)
34	136 137 138 139	**EK**((34-1)*4)XOR **EK**((34-6)*4)
35	140 141 142 143	**EK**((35-1)*4)XOR **EK**((35-6)*4)
36	144 145 146 147	**Sub Word**(**Rot Word**(**EK**((36-1)*4))) XOR **Rcon**((36/6)-1) XOR **EK**((36-6)*4)
37	148 149 150 151	**EK**((37-1)*4)XOR **EK**((37-6)*4)
38	152 153 154 155	**EK**((38-1)*4)XOR **EK**((38-6)*4)
39	156 157 158 159	**EK**((39-1)*4)XOR **EK**((39-6)*4)
40	160 161 162 163	**EK**((40-1)*4)XOR **EK**((40-6)*4)
41	164 165 166 167	**EK**((41-1)*4)XOR **EK**((41-6)*4)
42	168 169 170 171	**Sub Word**(**Rot Word**(**EK**((42-1)*4))) XOR **Rcon**((42/6)-1) XOR **EK**((42-6)*4)
43	172 173 174 175	**EK**((43-1)*4)XOR **EK**((43-6)*4)
44	176 177 178 179	**EK**((44-1)*4)XOR **EK**((44-6)*4)
45	180 181 182 183	**EK**((45-1)*4)XOR **EK**((45-6)*4)
46	184 185 186 187	**EK**((46-1)*4)XOR **EK**((46-6)*4)
47	188 189 190 191	**EK**((47-1)*4)XOR **EK**((47-6)*4)
48	192 193 194 195	**Sub Word**(**Rot Word**(**EK**((48-1)*4))) XOR **Rcon**((48/6)-1) XOR **EK**((48-6)*4)
49	196 197 198 199	**EK**((49-1)*4)XOR **EK**((49-6)*4)
50	200 201 202 203	**EK**((50-1)*4)XOR **EK**((50-6)*4)
51	204 205 206 207	**EK**((51-1)*4)XOR **EK**((51-6)*4)

- 32 byte Key Expansion

Each round (except rounds 0, 1, 2, 3, 4, 5, 6 and 7) will take the result of the previous round and produce a 4 byte result for the current round. Notice the first 8 rounds simply copy the total of 32 bytes of the key.

Table 14: 32-byte key expansion [9]

Round	Expanded Key Bytes	Function
0	0 1 2 3	K(0)
1	4 5 6 7	K(4)
2	8 9 10 11	K(8)
3	12 13 14 15	K(12)
4	16 17 18 19	K(16)
5	20 21 22 23	K(20)
6	24 25 26 27	K(24)
7	28 29 30 31	K(28)
8	32 33 34 35	Sub Word(Rot Word(EK((8-1)*4))) XOR Rcon((8/8)-1) XOR EK((8-8)*4)
9	36 37 38 39	EK((9-1)*4)XOR EK((9-8)*4)
10	40 41 42 43	EK((10-1)*4)XOR EK((10-8)*4)
11	44 45 46 47	EK((11-1)*4)XOR EK((11-8)*4)
12	48 49 50 51	Sub Word(EK((12-1)*4))XOR EK((12-8)*4)
13	52 53 54 55	EK((13-1)*4)XOR EK((13-8)*4)
14	56 57 58 59	EK((14-1)*4)XOR EK((14-8)*4)
15	60 61 62 63	EK((15-1)*4)XOR EK((15-8)*4)
16	64 65 66 67	Sub Word(Rot Word(EK((16-1)*4))) XOR Rcon((16/8)-1) XOR EK((16-8)*4)
17	68 69 70 71	EK((17-1)*4)XOR EK((17-8)*4)
18	72 73 74 75	EK((18-1)*4)XOR EK((18-8)*4)
19	76 77 78 79	EK((19-1)*4)XOR EK((19-8)*4)
20	80 81 82 83	Sub Word(EK((20-1)*4))XOR EK((20-8)*4)
21	84 85 86 87	EK((21-1)*4)XOR EK((21-8)*4)
22	88 89 90 91	EK((22-1)*4)XOR EK((22-8)*4)
23	92 93 94 95	EK((23-1)*4)XOR EK((23-8)*4)
24	96 97 98 99	Sub Word(Rot Word(EK((24-1)*4))) XOR Rcon((24/8)-1) XOR EK((24-8)*4)
25	100 101 102 103	EK((25-1)*4)XOR EK((25-8)*4)
26	104 105 106 107	EK((26-1)*4)XOR EK((26-8)*4)
27	108 109 110 111	EK((27-1)*4)XOR EK((27-8)*4)
28	112 113 114 115	Sub Word(EK((28-1)*4))XOR EK((28-8)*4)
29	116 117 118 119	EK((29-1)*4)XOR EK((29-8)*4)
30	120 121 122 123	EK((30-1)*4)XOR EK((30-8)*4)
31	124 125 126 127	EK((31-1)*4)XOR EK((31-8)*4)
32	128 129 130 131	Sub Word(Rot Word(EK((32-1)*4))) XOR Rcon((32/8)-1) XOR EK((32-8)*4)
33	132 133 134 135	EK((33-1)*4)XOR EK((33-8)*4)
34	136 137 138 139	EK((34-1)*4)XOR EK((34-8)*4)
35	140 141 142 143	EK((35-1)*4)XOR EK((35-8)*4)
36	144 145 146 147	Sub Word(EK((36-1)*4))XOR EK((36-8)*4)
37	148 149 150 151	EK((37-1)*4)XOR EK((37-8)*4)
38	152 153 154 155	EK((38-1)*4)XOR EK((38-8)*4)
39	156 157 158 159	EK((39-1)*4)XOR EK((39-8)*4)
40	160 161 162 163	Sub Word(Rot Word(EK((40-1)*4))) XOR Rcon((40/8)-1) XOR EK((40-8)*4)
41	164 165 166 167	EK((41-1)*4)XOR EK((41-8)*4)
42	168 169 170 171	EK((42-1)*4)XOR EK((42-8)*4)
43	172 173 174 175	EK((43-1)*4)XOR EK((43-8)*4)
44	176 177 178 179	Sub Word(EK((44-1)*4))XOR EK((44-8)*4)
45	180 181 182 183	EK((45-1)*4)XOR EK((45-8)*4)
46	184 185 186 187	EK((46-1)*4)XOR EK((46-8)*4)
47	188 189 190 191	EK((47-1)*4)XOR EK((47-8)*4)
48	192 193 194 195	Sub Word(Rot Word(EK((48-1)*4))) XOR Rcon((48/8)-1) XOR EK((48-8)*4)
49	196 197 198 199	EK((49-1)*4)XOR EK((49-8)*4)
50	200 201 202 203	EK((50-1)*4)XOR EK((50-8)*4)
51	204 205 206 207	EK((51-1)*4)XOR EK((51-8)*4)
52	208 209 210 211	Sub Word(EK((52-1)*4))XOR EK((52-8)*4)
53	212 213 214 215	EK((53-1)*4)XOR EK((53-8)*4)
54	216 217 218 219	EK((54-1)*4)XOR EK((54-8)*4)
55	220 221 222 223	EK((55-1)*4)XOR EK((55-8)*4)
56	234 225 226 227	Sub Word(Rot Word(EK((56-1)*4))) XOR Rcon((56/8)-1) XOR EK((56-8)*4)
57	228 229 230 231	EK((57-1)*4)XOR EK((57-8)*4)
58	232 233 234 235	EK((58-1)*4)XOR EK((58-8)*4)

4.3. Implementation details

The following functions are required by both encryption and decryption modules as these functions are required for key generation and some computational steps:

generateSubkeys

Input: byte[] key

Returns: byte[] tmp

Pseudo Code:

```
byte[][] tmp = new byte[Nb * (Nr + 1)][4]

int i = 0

while (i < Nk)

tmp[i][0] = key[i * 4]

tmp[i][1] = key[i * 4 + 1]

tmp[i][2] = key[i * 4 + 2]

tmp[i][3] = key[i * 4 + 3]

i++

i = Nk

while (i < Nb * (Nr + 1))

byte[] temp = new byte[4]

for(int k = 0;k<4;k++)

temp[k] = tmp[i-1][k]

if (i % Nk == 0)

temp = SubWord(rotateWord(temp))

temp[0] = (byte) (temp[0] ^ (Rcon[i / Nk] & 0xff))

else if (Nk > 6 && i % Nk == 4)
```

temp = SubWord(temp);

tmp[i] = xor_func(tmp[i - Nk], temp)

i++

return tmp

xor_func

Input: byte[] a, byte[] b

Returns: byte[] out

Pseudo Code:

byte[] out = new byte[a.length]

for (int i = 0; i < a.length; i++)

out[i] = (byte) (a[i] ^ b[i])

return out

SubWord

Input: byte[] in

Returns: byte[] tmp

Pseudo code:

byte[] tmp = new byte[in.length]

for (int i = 0; i < tmp.length; i++)

tmp[i] = (byte) (sbox[in[i] & 0x000000ff] & 0xff)

return tmp

rotateWord

Input: byte[] input

Returns: byte[] tmp

Pseudo code:

byte[] tmp = new byte[input.length]

tmp[0] = input[1]

tmp[1] = input[2]

tmp[2] = input[3]

tmp[3] = input[0]

return tmp

FFMul

Input: byte a, byte b

Output: byte r

Pseudo Code:

byte aa = a, bb = b, r = 0, t

while (aa != 0)

if ((aa & 1) != 0)

r = (byte) (r ^ bb)

t = (byte) (bb & 0x80)

bb = (byte) (bb << 1)

if (t != 0)

bb = (byte) (bb ^ 0x1b)

aa = (byte) ((aa & 0xff) >> 1)

return r

4.3.1. Encryption

The encryption algorithm has the following:

Constants - Nb = 4; Nk = key.length/4; Nr = Nk + 6; int lenght=0;

Inputs - byte[] in, byte[] key

The input text is first checked and is passes through byte padding sequence in order to make sure it contains sufficient number of bytes for encryption.

encryptBloc

Input: byte[] in

Returns: byte[] tmp

Pseudo code:

```
byte[] tmp = new byte[in.length]

byte[][] state = new byte[4][Nb]

for (int i = 0; i < in.length; i++)

state[i / 4][i % 4] = in[i%4*4+i/4]

state = AddRoundKey(state, w, 0)

for (int round = 1; round < Nr; round++)

state = SubBytes(state)

state = ShiftRows(state)

state = MixColumns(state)

state = AddRoundKey(state, w, round)

state = SubBytes(state)

state = ShiftRows(state)

state = AddRoundKey(state, w, Nr)
```

for (int i = 0; i < tmp.length; i++)

tmp[i%4*4+i/4] = state[i / 4][i%4]

return tmp

AddRoundKey

Input: byte[][] state, byte[][] w, int round

Output: byte[][] tmp

Pseudo Code:

byte[][] tmp = new byte[state.length][state[0].length]

for (int c = 0; c < Nb; c++)

for (int l = 0; l < 4; l++)

tmp[l][c] = (byte) (state[l][c] ^ w[round * Nb + c][l])

return tmp

SubBytes

Input: byte[][] state

Output: byte[][] tmp

Pseudo Code:

byte[][] tmp = new byte[state.length][state[0].length]

for (int row = 0; row < 4; row++)

for (int col = 0; col < Nb; col++)

tmp[row][col] = (byte) (sbox[(state[row][col] & 0x000000ff)] & 0xff)

return tmp

ShiftRows

Input: byte[][] state

Output: byte[][] state

Pseudo Code:

byte[] t = new byte[4]

for (int r = 1; r < 4; r++)

for (int c = 0; c < Nb; c++)

t[c] = state[r][(c + r) % Nb]

for (int c = 0; c < Nb; c++)

state[r][c] = t[c]

return state

MixColumns

Input: byte[][] s

Output: byte[][] tmp

Pseudo Code:

int[] sp = new int[4]

byte b02 = (byte)0x02, b03 = (byte)0x03

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

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

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

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

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

for (int i = 0; i < 4; i++)

s[i][c] = (byte)(sp[i])

return s

4.3.2. Decryption

The decryption algorithm has the following:

Constants - Nb = 4; Nk = key.length/4; Nr = Nk + 6; int lenght=0;

Inputs - byte[] in, byte[] key

The input cipher text is first decrypted and is then passes through byte padding sequence in order to make sure it contains sufficient number of bytes as the input plain text.

decryptBloc

Input: byte[][] in

Output: byte[] tmp

Pseudo Code:

byte[] tmp = new byte[in.length]

byte[][] state = new byte[4][Nb]

for (int i = 0; i < in.length; i++)

state[i / 4][i % 4] = in[i%4*4+i/4]

state = AddRoundKey(state, w, Nr)

for (int round = Nr-1; round >=1; round--)

state = InvSubBytes(state)

state = InvShiftRows(state)

state = AddRoundKey(state, w, round)

state = InvMixColumns(state)

state = InvSubBytes(state)

state = InvShiftRows(state)

```
state = AddRoundKey(state, w, 0)

for (int i = 0; i < tmp.length; i++)

tmp[i%4*4+i/4] = state[i / 4][i%4]

return tmp
```

InvSubBytes

Input: byte[][] state

Output: byte[][] state

Pseudo Code:

```
for (int row = 0; row < 4; row++)

for (int col = 0; col < Nb; col++)

state[row][col] = (byte)(inv_sbox[(state[row][col] & 0x000000ff)]&0xff)

return state
```

InvShiftRows

Input: byte[][] state

Output: byte[][] state

Pseudo Code:

```
byte[] t = new byte[4]

for (int r = 1; r < 4; r++)

for (int c = 0; c < Nb; c++)

t[(c + r)%Nb] = state[r][c]

for (int c = 0; c < Nb; c++)

state[r][c] = t[c]
```

return state

InvMixColumns

Input: byte[][] s

Output: byte[][] state

Pseudo Code:

```
int[] sp = new int[4]

byte b02 = (byte)0x0e, b03 = (byte)0x0b, b04 = (byte)0x0d, b05 = (byte)0x09

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

sp[0] = FFMul(b02, s[0][c]) ^ FFMul(b03, s[1][c]) ^ FFMul(b04,s[2][c]) ^ FFMul(b05,s[3][c])

sp[1] = FFMul(b05, s[0][c]) ^ FFMul(b02, s[1][c]) ^ FFMul(b03,s[2][c]) ^ FFMul(b04,s[3][c])

sp[2] = FFMul(b04, s[0][c]) ^ FFMul(b05, s[1][c]) ^ FFMul(b02,s[2][c]) ^ FFMul(b03,s[3][c])

sp[3] = FFMul(b03, s[0][c]) ^ FFMul(b04, s[1][c]) ^ FFMul(b05,s[2][c]) ^ FFMul(b02,s[3][c])

for (int i = 0; i < 4; i++)

s[i][c] = (byte)(sp[i])

return s
```

4.4. Performance Analysis

The performance parameters considered for enhancement are calculated.

The time taken for encryption and decryption in case of conventional and improved algorithm are as follows:

For conventional AES:

$T\_encryption_c = 5ms$ – Time taken for encrypt

$T\_decryption_c = 3ms$ – Time taken to decrypt

For modified AES:

$T\_encryption_p = 3ms$ – Time taken to encrypt

$T\_decryption_p = 2ms$ – Time taken to decrypt

The following are the throughput values of encryption and decryption time for conventional AES algorithm.

$$\text{Throughput}_{encryption} = \frac{Block\ Size}{Time\ Taken_C}$$

$$= \frac{128}{5}$$

$$= 25.6$$

$$\text{Throughput}_{decryption} = \frac{Block\ Size}{Time\ Taken_C}$$

$$= \frac{128}{3}$$

$$= 42.67$$

The Throughput is given in the general form of completions per unit of time, the common throughput metric is instructions per cycle, In case of AES the throughput depends on size of block as well as time taken for encryption/decryption given by:

$$T = \frac{block\ size}{t}$$

Where,

T - Throughput

t - Time taken to encrypt/decrypt

The following are the throughput values of encryption and decryption time for modified AES algorithm.

$$\text{Throughput}_{p\_encryption} = \frac{Block\ Size}{Time\ Taken_p}$$

$$= \frac{128}{3}$$

$$= 42.67$$

$$\text{Throughput}_{p\_decrcryption} = \frac{Block\ Size}{Time\ Taken_p}$$

$$= \frac{128}{2}$$

$$= 64$$

Thus, the speedup achieved is given by the following relation:

$$S = \frac{T_{old}}{T_{new}}$$

Where,

S is the resultant speedup.

T_{old} is the old execution time, i.e., without the improvement.

T_{new} is the new execution time, i.e., with the improvement.

$$Speedup_{encryption} = \frac{Time\ Taken_c}{Time\ Taken_p}$$

$$= \frac{5}{3}$$

$$= 1.67$$

$$Speedup_{decryption} = \frac{Time\ Taken_c}{Time\ Taken_p}$$

$$= \frac{3}{2}$$

= 1.5

Chapter 5

Conclusion and Future Work

The Advanced Encryption Technique was implemented successfully using Java with an increased throughput and thus speedup. Various data messages were encrypted using different keys and varying key sizes. The original data was properly retrieved via decryption of the cipher text. The modifications brought about in the code was tested and proved to be accurately encrypting and decrypting the data messages with even higher security and immunity against the unauthorized users.

The limitations with this AES algorithm are: the successful attack against AES data encryption has been side channel attacks, which don't attack the actual AES cipher text, rather than its implementation. Since it drives on blocks of 200 bits it requires more processing for large data.

Further enhancement to this project can be testing the security of the modified algorithm against modern attacks. Linear and differential cryptanalysis can be used to analyze the level of security provided by the improved algorithm.

References

1. William Stallings, Cryptography and Network Security, Fifth Edition, Pearson Education, 2013.
2. Neal R. Wagner, "The Laws of Cryptography with Java Code", 2003 Edition.
3. Edward Chu, Paul Kim, Frank Liu, Jason Sharma and Jeffrey Yu, "The selection of Advanced Encryption Standard", MIT 6, 933J, fall 2000.
4. Douglas Selent, "Advanced Encryption Standard", Rivier Academic Journal, Vol. 6, No. 2, pp. 1-14 fall 2010.
5. Shaaban Sahmoud, Wisam Elmasry and Shadi Abudalfa, "Enhancement the Security of AES Against Modern Attacks by Using Variable Key Block Cipher", International Arab Journal of e -Technology, Vol. 3, No. 1, pp. 17-26, Jan 2013.
6. The Deguang Le, Jinyi Chang, Xingdou Gou, Ankang Zhang and Conglan Lu, "Parallel AES Algorithm for Fast Data Encryption on GPU", 2nd International Conference on Computer Engineering and Technology, Vol. 6, pp. 1-6, 2010.
7. Ritu Pahal and Vikas kumar, "Efficient Implementation of AES", International Journal of Advanced Research in Computer Science and Software Engineering, Vol. 3, Issue 7, pp. 290-295, Jul 2013.
8. Vishal Pachori, Gunjan Ansari and Neha Chaudhary, "Improved Performance of Advance Encryption Standard using Parallel Computing", International Journal of Engineering Research and Applications, Vol. 2, Issue 1, pp. 967-971, Jan-Feb 2012.
9. Adam Berent, "Advanced Encryption Standard by Example", documentation for ABI Software Development.
10. Kundankumar Rameshwar Saraf, Vishal Prakash Jagtap and Amit Kumar Mishra, "Text and Image Encryption Decryption Using Advanced Encryption Standard", International Journal of Emerging Trends & Technology in Computer Science, Vol. 3, Issue 3, pp.118-126, May-Jun 2014.
11. Alan Kaminsky, Michael Kurdziel and Stanisław Radziszowski1, "An Overview of Cryptanalysis Research for the Advanced Encryption Standard", Military Communications Conference, pp. 1310-1316, Oct-Nov 2010.

12. M.Pitchaiah, Philemon Daniel and Praveen, "Implementation of Advanced Encryption Standard Algorithm", International Journal of Scientific & Engineering Research Vol 3, Issue 3, pp. 1-6, Mar 2012.
13. Mg Suresh and Dr.Nataraj K.R., "Area Optimized and Pipelined FPGA Implementation of AES Encryption and Decryption", International Journal of Computational Engineering Research, Vol. 2, Issue 7, pp. 133-139, Nov 2012.
14. M. Shahraki, "Implementation Aspects of Rijndael Encryption Algorithm", Lecture Series on Computer and Computational Sciences, Vol. 2, pp. 166-169, 2005.
15. Vandana C. Koradia, "Modification in Advanced Encryption Standard", Journal of Information, Knowledge and Research in Computer Engineering", Vol. 2, Issue 2, pp. 356-358, 2013.
16. Amit Kumar Mangal, "Multithreaded Java Applications Performance Improvement", International Journal of Advanced Research in Computer Science and Software Engineering, Vol. 3, Issue 3, pp. 47-50, March 2013.

Appendix

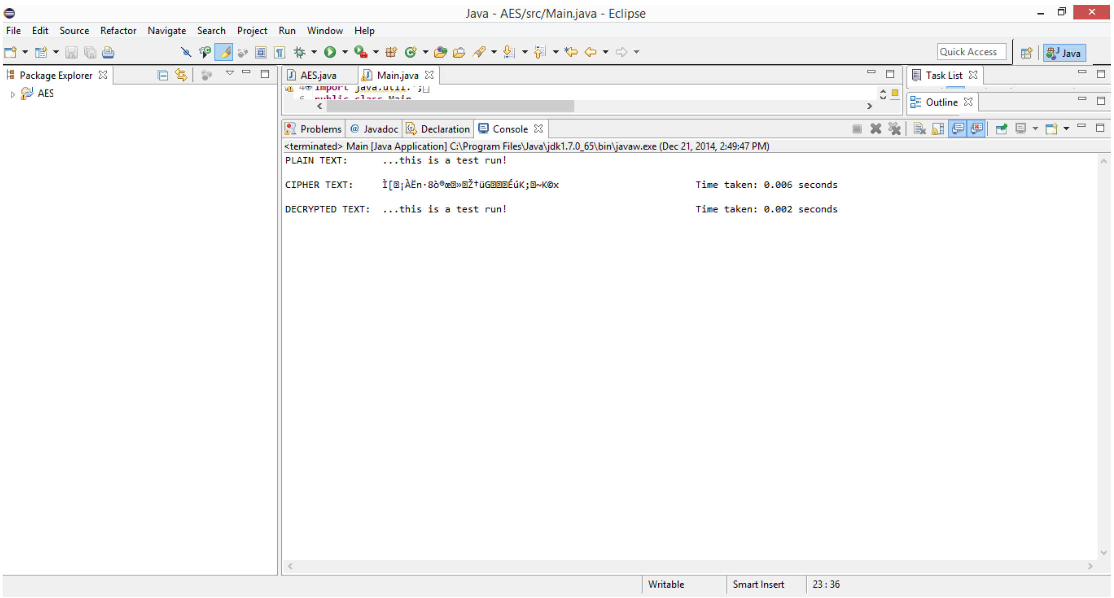

Figure 9: Screen capture of Console

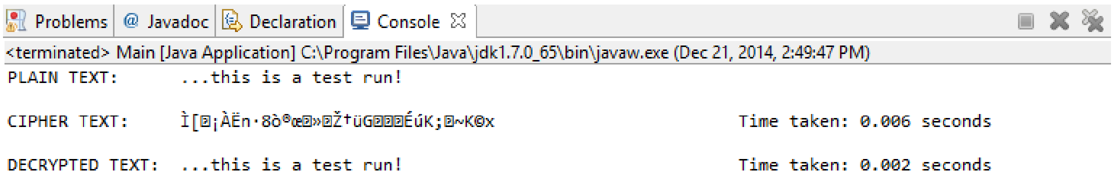

Figure 10: Output of code