Implementation and Analysis of the Advanced Encryption Standard (AES) Algorithm in Different Memory Configurations

Christopher Feldwick

BSc (Hons) in Mathematics and Computing

May 2005

Implementation and Analysis of the Advanced Encryption Standard (AES) Algorithm in Different Memory Configurations

Submitted by Christopher Feldwick

Declaration

ABSTRACT

In 1997 the US National Institute for Standards and Technology unveiled Rijndael as the new Advanced Encryption Standard (AES) to be used throughout the US government to encrypt sensitive (but not classified) data. The specification for the algorithm has left enough room to implement the various data transformations using a broad range of techniques.

These techniques are carefully researched and a combination of suitable ones are used to create two separate AES implementations, one optimised to use as little memory space as possible and the other optimised to increase execution speed.

The implementations are extensively tested against precompiled test values and analysis of the execution times is carried out. This analysis is then used to identify the different advantages amongst the two implementations and therefore suggest the circumstances where each would be best suited.

The dissertation then concludes by discussing the possible extensions to the work carried out in this project including the ways in which the implementations can be combined to create hybrid solutions for specific tasks.

ACKNOWLEDGEMENTS

I would like to thank Dr R. Bradford for coming up with the idea for such an interesting and challenging project.

I would also like to thank my project supervisor Prof J. Davenport for introducing me to a new and original direction to take the project in.

Finally I would like to give thanks to the friends and family members who have shown great support during the development of this project.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

Encryption is always going to be necessary for maintaining privacy in our personal communications. In fact it was created for that exact reason. However people are always finding new ways of “breaking” encryption and therefore leaving these communications open to being read or possibly tampered with by a third party.

In 1997 The Unites States National Institute of Standards and Technology (NIST) asked the public, including academics and professionals, to submit new cryptography algorithms as possible candidates to become the new Advanced Encryption Standard (AES). The previous standard called the Data Encryption Standard (DES) was in need of replacement by a new algorithm which was to be used throughout the US government to encrypt sensitive (but not classified) information.

There were 3 main candidates for the new standard namely, Rijndael, Serpent, 2fish, RC6 and MARS. In November 2001 Rijndael was chosen as the new AES algorithm. Rijndael can encrypt using 128, 192 and 256 bit cipher keys which are applied, using the algorithm, to data blocks of 128bits to perform the required transformation.

The aim of this dissertation is to produce two implementations of the AES encryption algorithm for use in two very different circumstances. The first of these implementations is designed for use in a conventional home computer system. The main requirement for this version is to maximise the speed at which the encryption can be achieved at the cost of memory usage. The second implementation has the reverse requirements of the first. The solution aims to minimise usage of memory space at the cost of speed of execution. This solution would be useful in small devices such as mobile phones, personal digital assistants (PDA’s) and possibly smart chips used in credit and debit cards.

Many of the operations required during the encryption process are mathematically complex, and require considerable computation time to complete. Some of these operations are needed hundreds of times in order to complete the encryption of a single 128bit block of data. However, it is possible to create lookup tables for some of these complex operations when the algorithm commences. This requires more memory space to hold the tables but considerably reduces the computational time taken by the complex operations. This will be the technique used by the first implementation to increase speed at the cost of memory, whereas in contrast, the second implementation will carry out as many of these operations as possible "on-the-fly" to reduce memory usage.

Unfortunately, initial creation and initialisation of lookup tables takes up computing time, however, once this process has been completed the same tables can be used to encrypt large amounts of data without having to reinitialise them. The low-memory solution will not have this added start-up time required to create the tables, however, the slow operations will have to be used more and more times as the amount of data to be encrypted increases.

This dissertation will analyse the amount of time taken by each of the solutions when the size of the input to be encrypted is varied. From this analysis it is hoped that conclusions can be drawn in relation to the point at which the first implementation, where the lookup tables have to be created and the encryption carried out, becomes faster than encrypting using the second implementation where calculations are done on the fly without the need for lookup tables.

The way that AES has been specified has left large scope for programmers to be able to implement the algorithm in a way they feel is necessary for their problem. This has lead to there being a number of different implementations publicly available which have been adapted to specific problems. For this reason it maybe useful to use a combination of these implementations, as well as new ideas, to create a unique solution which satisfies the needs of this dissertation. Some of these publicly available implementations are studied on the following pages (chapter 2). Following this, an explanation of the mathematical background required to understand the algorithm is given along with an overview of the algorithm itself. Details of how this will be implemented, including how it will differ depending on the implementation type will then follow (chapters 3 and 4).

Chapter 2

Literature Review

2.1 Introduction

Firstly, a look at the history and background of encryption should produce a clearer understanding for the need of security and how implementations such as the one being produced can help with this need.

The next part of this review will look at the more specific encryption technique known as the Advanced Encryption Standard (AES) and explain why it is used, how it works and how it can be implemented. Following this, the investigation must be expanded to search for solutions for the problem at hand. This involves looking for possible optimisation techniques and how they can be incorporated into the solution as well as highlighting the differences needed between the two implementations.

In addition, this review will be considering other factors which will contribute to the overall performance of the final implementations. These factors are likely to include the choice of programming language since this can have a large effect of the overall efficiency of algorithms.

Finally, due to the rapid advances in the technology of encryption and cryptanalysis it is necessary to examine the likelihood of the dissertation findings being obsolete shortly after they are produced. This has been known to happen in the past where weaknesses are found in other encryption systems. If this was the case then it may be necessary to use another encryption standard.

2.2 Background of Encryption

Firstly and possibly most importantly of all, it must be made clear why any research into computer security is necessary, since without this necessity for confidentiality, there is no point in this, or any other pieces of work in this area. Seberry and Pieprzyk (1989) explain,

> *"Security of information results from the need for private transmission of both military and diplomatic messages. This need is as old as civilization itself."*

They also go on to recognise that, in general, security in computer systems and networks consists of three components:

- Security of Computing Centre(s);
- Security of terminals;
- And security of communication channels.

In this project it is the latter that we are most concerned with since that is the main goal of encryption. Cryptography, the science of secret writing, has been used by individuals and nations for centuries to prevent access to information in messages (Kahn 1967, cited by Lane 1985 p.43) and is the basis for encrypting messages across communication lines

The general objective for encryption is not to protect the communications as such but rather their content (Becket 1988). This is due to it being extremely difficult to stop communication interceptions. By using encryption, even if there is a transmission interception, the content of the communication is kept secret. This is extremely important in communication channels such as the Internet where many networks are connected together using broadcast mediums (such as Ethernet) and so data interception is a real and active problem.

One of the first and more primitive methods of encryption was substitution. This simply meant that every letter in a message was substituted for a different one. Once the message was delivered, the recipient would know how to change the letters back, and therefore reveal the original message. Since then a number of more advanced methods have been invented but many of them share a common weakness. Until more recently, all encryption techniques were linear transformations. That is, the relationship between the plaintext (the original message) and the ciphertext (the encrypted message) was linear. Becket (1988) tells us,

> *"Given, say, two examples of ciphertext encrypted with the same key, or a ciphertext with known or suspected plaintext, a cryptanalyst can break the system with relative ease despite its overall strength."*

For this reason linear encryption is now rarely used as the primary method for strong encryption.

In the 1970's the National Bureau of Standards (NBS) advertised for submission of possible candidates for a new encryption standard. The adopted method was proposed by IBM. It was known as the Data Encryption Standard (DES) and was published in 1977. The United States government used it to protect data they regarded as sensitive.

"For the mathematical point of view, the DES can be regarded as a simple cipher with an alphabet of 2^{64} letters; that is, the message is broken up into blocks of 64 bits, and each bit regarded as a letter"

(Welsh 1988)

DES encryption was subject to strong criticisms for its short key length and it has been often argued that attacks via "brute-force" (independent of the encryption algorithm structure) would break the encryption fairly quickly (Biham and Knudsen 1998). In fact by 1993 a machine had been developed which would cost a total of $1million that would find a key for the encryption in an average of 3.5 hours (Wiener 1996). It has been estimated that with today's technology it would be possible to find the key in an average of approximately 35mins (Wiener 1996).

For this reason it was decided that a new encryption standard should be introduced. This is where the Advanced Encryption Standard (AES) algorithm was invented.

2.3 Rijndael Encryption

2.3.1 What is Rijndael?

In 1997 The Unites states National Institute of Standards and Technology (NIST, formally NBS) asked the public, including academics and professionals, to submit new cryptography algorithms as possible candidates to become the New Advanced Encryption Standard (AES) to replace the out dated DES algorithm for use with sensitive, but not confidential government data.

There were 5 main candidates for the new standard namely, Rijndael, Serpent, 2fish, RC6 and MARS. In November 2001 Rijndael was chosen as the new AES standard. In performance comparison studies carried out on all five finalists, RIJNDAEL proved to be one of the fastest and most efficient algorithms. It is also implemented on a wide range of platforms and is extendable to different key and block lengths. (Standaert et al. 2003)

The following extract is taken from the submission paper written by the designers of the Rijndael algorithm and explains the reasons why they believed it was the best candidate.

- **Security:** *Rijndael has the same objective security level as the other finalists, and can easily be implemented in a secure way.*

- **Efficiency:** *Rijndael has a large "performance margin" compared to the other candidates.*
- **Design philosophy:** *The clear design has many advantages: easy implementable on a wide range of platforms, easy to get confidence in the claimed security level.*
- **Extensions:** Rijndael *is easily extendable to other key and block lengths*.

(Daemen and Rijmen 2000)

2.3.2 Breakdown of the Algorithm

The overall structure of the Rijndael algorithm is very straight forward and has often be praised for its simplicity. It is constructed from a repeated number of grouped operations called rounds (the number depends on the block length). The algorithm takes an encryption key of 128, 192 or 256 bit length as an input and a data block length of 128bits (The size of the blocks when the message has been split up).

The algorithm treats the input data as a matrix of bytes. A 4x4 matrix holds the input message in its individual bytes and the transformations in the algorithm are then acted upon this matrix. The “current state” is the 4x4 matrix after it has gone through a number of these transformations.

Current research in cryptanalysis shows that the resistance of the Rijndael algorithm to cryptanalytic attacks increases with the number of rounds there are (Daemen and Rijmen 2002). The choice for the number of rounds was decided by taking the number of rounds required to make short-cut attacks ineffective and then adding a number of rounds as a safety margin. The chosen number of rounds is different depending on the key length chosen:

- 10 rounds with a 128bit key length
- 12 rounds with a 192bit key length
- 14 rounds with a 256bit key length

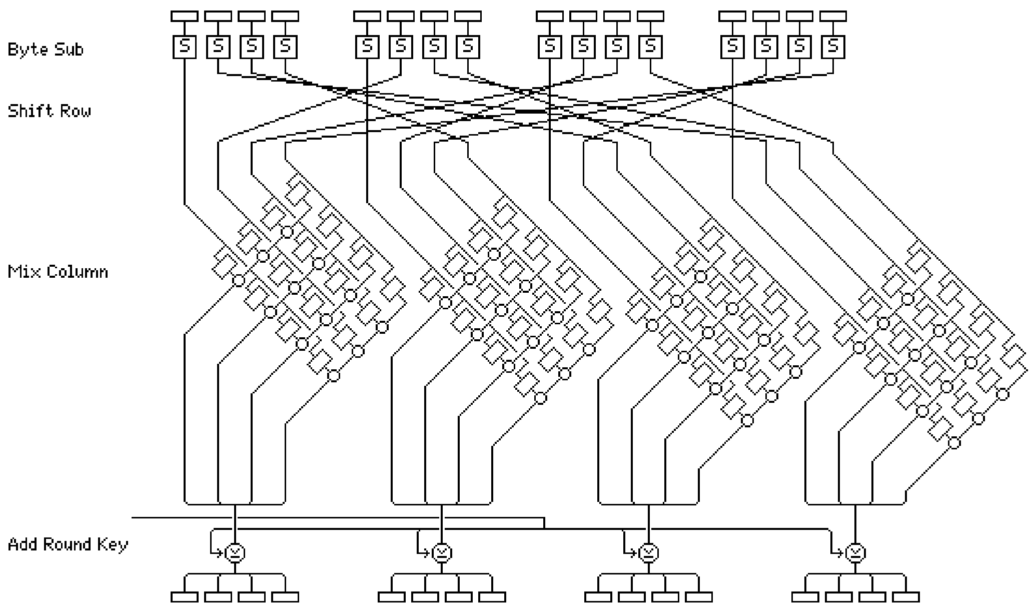
(url: TheAES)

The algorithm can be split into four separate functions which act on the “current state” These are explained in more detail later:

- Byte Substitution
- Shift Rows
- Mix Columns
- Add Round Key

The algorithm starts with an Add Round Key function and is followed by the repeated rounds which are constructed in the following order: Byte Substitution, Shift Row, Mix Column then Add Round Key. This is repeated for all the rounds except the last where the Mix Columns operation is omitted (Daemen and Rijmen 2002).

A diagram of the process can be seen below:



(url: TheAES)

The following sections give a quick overview of the roles of each of the functions. These areas will of course be explained in full detail later in the dissertation.

Byte Substitution:

The byte substitution is a non-linear transformation which works independently on each byte of the current state. This can be achieved by using a substitution table (s-box) and swapping each byte for the equivalent in the s-box. (Anon 2001 p.19) or carrying out the transformation separately byte-by-byte

The s-box is invertible and is constructed from two transformations. Firstly a mathematical inverse of the byte is calculated and then followed by a linear transformation. Since the inversion is non-linear, the whole process is therefore non-linear as well.

The Byte Substitution function is the only non-linear transformation in the algorithm (Daemen and Rijmen 2002). Without this, the algorithm would be completely linear and therefore, as explained earlier, easier to break.

Shift Rows:

This is a relatively straight forward process since the “state” is being held in a matrix. The bytes in the last 3 rows are cyclically shifted over a number of different offsets. The first row is left unchanged.

Mix Columns:

This function acts on the state matrix column-by-column. For this it treats each column as a four termed polynomial (explained further later in the project). This polynomial is then multiplied by another predetermined polynomial modulo $x^4 + 1$. This results in a new four termed polynomial which can then be translated back into a column in the new state. (Anon 2001 p.22)

The design criteria for this choice of function took the following areas into consideration:

1. **Dimensions**. *The transformation is a bricklayer layer transformation operating on four byte columns*.
2. **Linearity**. *The transformation is preferably linear over GF(2).*
3. **Diffusion**. *The transformation has to have relevant diffusion power*.
4. **Performance on 8-bit processors**. *The performance of the transformation on 8-bit processors has to be high.*

(Daemen and Rijmen 2002)

Add Round Key:

The Add Round Key transformation is the final step in the round and is the most straightforward. This function makes use of the key and therefore without it, the algorithm would be useless.

The process simply calculates a bitwise XOR with the subkey depending on the current round number. The subkey is generated from the original input key using an expanding algorithm and is dependant on the round that the algorithm is currently on (url: TheAES).

2.4 Optimisation Techniques

This project is not only concerned about the general implementation of the algorithm but also how it can be optimised for use in low-memory systems.

The AES has been subject to many different forms of optimisation by many different people. This includes hardware optimisation as shown by Standaert et al. (2003) and by Rijmen (2000). Other areas which are of interest to this project are choice of programming language and implementation of the S-box in the algorithm.

To produce the best performance out of the algorithm it is necessary to initialise the S-box at the start of the process and then use it in each round to make the byte substitutions, however this requires an array of 256k. A small as this may seem, it still may be necessary to release this memory in a low-memory system. It has been shown in Gladman (2003) that it is possible to achieve the substitution on-the-fly and without the need for the s-box. Of course this is likely to reduce the performance considerably, however this compromise may have to be made in order for very low-memory systems to be able to implement AES. This is backed up by the makers of the algorithm themselves who explain:

"The Round Key addition, ByteSub and RowShift can be efficiently combined and executed serially per State byte. Indexing overhead is minimised by explicitly coding the operation for every state byte."

(Daemen and Rijmen 1999)

Another optimisation need is for 8-bit processing. Because of the way Rijndael was designed it easily caters for these situations:

"The Rijndael cipher is suited to be implemented efficiently on a wide range of processors and in dedicated hardware…..
On an 8-bit processor, Rijndael can be programmed by simply implementing the different component transformations. This is straightforward for RowShift and for the Round Key addition. The implementation of ByteSub requires a table of 256 bytes."

(Daemen and Rijmen 1999)

However, as explained before a compromise between memory usage and speed needs to be considered.

There are many different optimisations and implementations available to the public, however, due to the massive number of possible applications for the algorithm it is often very hard to find an exact match for the problem at hand.

For this reason it is a good idea to design a new implementation from the start which can be optimised for the exact problem at hand.

2.5 The Future of AES

It is very important to look at the future of the Rijndael algorithm since there is no point implementing an encryption algorithm with known floors. The eventual downfall of the DES was the key length as the exhaustive list of possible keys was not big enough and so as technology advanced it became easier to test all the keys quickly.

The main change from DES to AES is well motivated from a security point of view (url: crypto). Ferguson et al. (2002) has done lots of cryptanalysis on the algorithm and found techniques to attempt to break up to 6 rounds of the 128-bit version however that still leaves a security margin of 4 rounds which are untouched. Another possibility that remains is that of “brute force”. This requires testing every possible key to find a match. This is a completely unusable tactic when using 128 bit keys since this equates to $3.40282367 \times 10^{38}$ possible keys. This is an immense amount of keys and there is not a computer in the foreseeable future which could ever process that number of keys.

In addition to this, the specification for Rijndael means that the number of rounds is a parameter which can be increased further without any need for additional specifications. This means that if a security need was extremely great then an increase in rounds is very easy to implement. (Daemen & Rijmen 2000). Also if a successful attack on the algorithm was published then the number of rounds could be increased to counteract it.

Out of the 5 AES finalists, Rijndael is the only one which allows for different size block lengths as an extension, namely 192 and 256 bit blocks. (Daemen & Rijmen 2000).

There is a great deal of confidence in the Rijndael algorithm due to the reasons specified above.

2.6 Other Previous Work

As mentioned earlier, there has been a lot of research into AES. Since the announcement of Rijndael as the new standard, many different implementations have appeared on the internet, each being modified for specific purposes. These wide varieties of examples allow new implementations to pick and choose parts of others in order to create a new version optimised for a completely different problem.

An example of an existing implementation is displayed in Buchholz (2001) where an implementation in Matlab is demonstrated. Since the AES algorithm generally acts upon the "state" matrix, using Matlab makes it very clear how the mathematics of the algorithms works as the Matlab programming language is based on matrix manipulation. Understanding how the mathematics of the algorithm works is essential in producing a good solution and for that reason this implementation is extremely useful. However, the author explains that there is little optimisation in the code:

"Even though this implementation is fully operational, (i. e. it can be utilized to encrypt arbitrarily chosen plaintext into ciphertext and vice versa), the main optimization parameter of this implementation has not been execution speed but understandability."

(Buchholz 2001)

One such example of where optimisation would normally have been used in the code is when calculating an inverse of a byte (explained further later). In the Matlab code the author's method for calculating an inverse involves trying all possibilities until the correct one is found. This process is slow and there exist alternative solutions which are much faster and use similar amounts of memory space.

One person who has done a large amount of research into the AES algorithm is Dr Brian Gladman. An example of this is in Gladman (2003) where the author has updated the original algorithm specification and included many techniques which can be used to increase the efficiency of the encryption process. Such techniques include using logarithm and anti-logarithm tables to calculate multiplication of bytes and an efficient algorithm to complete the mix columns operation of the encryption. These techniques are likely to be incorporated into the new implementation.

Possibly the most mathematically complex operation in the algorithm is finding the multiplicative inverse of a byte. This operation is used in both the sub byte and the key expansion routines of the encryption algorithm. Ward and Moltneo (2003) explore the different possible methods for solving this problem. Although the paper is specifically meant for hardware implementations of the algorithm, some of the methods can be translated into a software based solution. One such example is the Extended Euclidean Algorithm. This routine finds the greatest common divisor between two inputs and can be used to find inverses of numbers modulus an irreducible constant. Because of the special way bytes are multiplied in Rijndael, the Extended Euclidean Algorithm can be used to find inverses of bytes. (This will be explained further in chapter 4). Hankerson et al. (2003, Ch. 2) extends on these findings by giving the Binary Extended Euclidean Algorithm pseudocode for use in finding inverses in finite fields. This specific algorithm can be used to find the inverses of bytes as necessary in Rijndael and is regarded by many as the best solution to the problem.

2.7 Choice of Programming Language

The final consideration to make before any implementation details can be considered is the choice of programming language. This can have a large effect over the performance of the algorithm which is of high importance to this dissertation.

Since the coding of the solution may be quite complex it will be an advantage to use a language which I am already competent in. These include C, Matlab and Java. The advantage of using Matlab would be that the algorithm can work directly on matrices which Matlab can maintain and therefore allow for a more natural, and easier to understand solution. However, the downside to Matlab is the performance as it cannot produce computational speeds close to C. Since this is one of the main objectives for one of the solutions, Matlab will not be used.

If Java was to be used then the application would be able to run on multiple operating systems and processors. As advantageous as this is, it does result in loss of performance again compared to C. Also there is little or no scope for using the object-orientated advantage of Java for the problem at hand. For these reasons I have chosen C as the programming language for the implementations.

Chapter 3

Mathematical Background

As explained earlier, the Rijndael algorithm acts upon a 4x4 matrix with each matrix entry holding one byte of data. Initially the input string to be encrypted is split into separate bytes and then placed into the matrix in a specified order.

Once in the form of a matrix, the algorithm applies several different mathematical operations to the bytes with the outputs of these operations also being bytes. Since a single byte of data can only represent a finite number of elements (e.g. 1 byte can represent 256 different decimal numbers) it means that these operations are acting on a finite field. These types of fields are known as the Galois Fields. This particular one is denoted GF (2^8) as there are 2^8 different elements.

In order to use this field it is necessary to fully define some simple arithmetic operations. In order to define them a method of representing a byte mathematically must be chosen. There are a number of different ways to represent the field GF (2^8) however; all of them are in fact isomorphic. In a mathematical sense this means they are all essentially equivalent.

3.1 Polynomial Representation of GF(2^8)

The designers of the Rijndael algorithm chose to use the most common representation for elements in GF (2^8). This method regards each byte as a polynomial of order 7.

Each byte can be written as a series of 8 bits. These bits represent the coefficients of the polynomial of order 7. For example:

A byte $b_7b_6b_5b_4b_3b_2b_1b_0$ is represented by the polynomial:

$$\begin{tikzcd}
b_7 \arrow[d] & b_6 \arrow[d] & b_5 \arrow[d] & b_4 \arrow[d] & b_3 \arrow[d] & b_2 \arrow[d] & b_1 \arrow[d] & b_0 \arrow[d] \\
b_7x^7 + & b_6x^6 + & b_5x^5 + & b_4x^4 + & b_3x^3 + & b_2x^2 + & b_1x + & b_0
\end{tikzcd}$$

Therefore the hexadecimal number $2F_{16}$ which is equal to 00101111_2 would be the equation:
$x^5 + x^3 + x^2 + x + 1$

3.2 Arithmetic Operations in GF(2^8)

3.2.1 Addition

Now that the representation of bytes is clear it is possible to introduce some of the basic mathematical operations that can be applied to these bytes.

The standard way to add polynomials is to sum the coefficients of like powers. However, for the polynomial representation work, the coefficients can only ever be 1 or 0. In order to achieve this, when summing the coefficients, the result is given modulo 2. (i.e. 1 + 1 = 0).

For example adding 57_{16} and 83_{16}:

$$(x^6 + x^4 + x^2 + x + 1) + (x^7 + x^5 + x + 1)$$

$$= x^7 + x^6 + x^5 + x^4 + x^2 + x + x + 1 + 1$$
$$= x^7 + x^6 + x^5 + x^4 + x^2$$

The process of adding these two bytes is actually equal to a bitwise XOR of the bytes concerned. This means that addition in GF (2^8) can be carried out extremely quickly on the computer.

3.2.2 Subtraction

The subtraction of elements in GF (2^8) works using the same rules as with additions. The coefficients of the polynomial to be subtracted are taken away from the coefficients of the equivalent power. Again since only the digits 0 and 1 can be coefficients we have to set

$$0 - 1 = 1\ . \qquad \textbf{\textit{(3.1)}}$$

Bit A	**Bit B**	**A XOR B**	**A + B**	**A - B**
0	0	0	0	0
1	0	1	1	1
0	1	1	1	1
1	1	0	0	0

Table 1 – The 4 possible results of an XOR, addition and subtraction of the polynomial coefficients.

As you can see from the table above, the process of adding and subtracting the coefficients gives exactly the same result as an XOR. This means that to accomplish an addition or subtraction of elements in GF (2^8), a bitwise XOR of the byte is all that is required.

3.2.3 Multiplication

Multiplication using the polynomial representation of GF (2^8) starts to become more complex than the previous two operations shown above. Using the standard multiplication of two polynomials of degree 7 creates problems. There is no guarantee that the resulting polynomial will be of degree 7 or less meaning that it will no longer be able to represent a single byte. In fact it is possible to have a polynomial of degree 14 since the maximum degree of a product of polynomials is the product of the degrees.

To counteract this situation the result of multiplication in GF (2^8) is defined as the multiplication of the polynomials modulo a fixed irreducible polynomial, m(x). This means that m(x) has no other divisors other than itself and 1 (The polynomial equivalent of a prime number). Unfortunately, unlike with additions, there is no quick bit-level operation for multiplication.

In the Rijndael specification m(x) is given as

$$m(x) = x^8 + x^4 + x^3 + x + 1. \qquad \textbf{(3.2)}$$

Below is an example of multiplication in GF (2^8) taken from Daemen and Rijmen (1999).

Example: ‘57’ • ‘83’ = ‘C1’, or:

$$\begin{aligned}(x^6 + x^4 + x^2 + x + 1)\,(x^7 + x + 1) \quad &= \quad x^{13} + x^{11} + x^9 + x^8 + x^7 + \\ &\qquad x^7 + x^5 + x^3 + x^2 + x + \\ &\qquad x^6 + x^4 + x^2 + x + 1 \\ &= \quad x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1\end{aligned}$$

$$\begin{aligned}x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1 \text{ modulo } x^8 + x^4 + x^3 + x + 1 \\ = \quad x^7 + x^6 + 1\end{aligned}$$

Figure 1 – Example of Multiplication in GF (2^8)

3.3 Multiplicative Inverses in GF(2^8)

Because m(x) is irreducible, the greatest common divisor of m(x) and any other polynomial is always equal to 1. This feature allows us to use the Extended Euclidean Algorithm to calculate multiplicative inverses of elements in GF (2^8) as follows.

Given *b(x)*, a byte in binary polynomial representation, and *m(x)* as above the Extended Euclidean Algorithm finds *a(x)* and *c(x)* such that:

$$b(x)a(x) + c(x)m(x) = \mathrm{gcd}(b(x), m(x))\ . \qquad \textbf{(3.3)}$$

Since *m(x)* is irreducible, *gcd(b(x), m(x))* must equal 1. Therefore

$$\begin{aligned} & b(x)a(x) + c(x)m(x) = 1 \\ => \quad & a(x)b(x) \bmod m(x) = 1 \\ => \quad & b^{-1}(x) = a(x) \quad . \qquad \textbf{(3.4)} \end{aligned}$$

Equation (2) shows how the polynomial *a(x)* found by the algorithm actually is the multiplicative inverse of a byte *b(x).* As we are working in a mathematical field we automatically know that each element of the field (except for 0) has an inverse.

3.4 Polynomials with Coefficients in GF(2^8)

Within the encryption algorithm there exist operations which act upon 4-byte vectors. In Rijndael these vectors are represented as a polynomial of degree 3 with the coefficients being elements of GF (2^8) unlike the coefficients earlier which were elements of GF (2).

For example, if the 4 bytes in hexadecimal are {01}, {02}, {03}, {04} then this is represented by the polynomial

$$\{01\}x^3 + \{02\}x^2 + \{03\}x + \{04\}$$

In order to use this representation it is again necessary to define the two main arithmetic functions, addition and multiplication.

3.4.1 Addition

As all the coefficients are elements of GF (2^8) the addition of these new polynomials is straightforward. It amounts to an XOR between the equivalent coefficients in the polynomials, which in turn is equivalent to a bitwise XOR of the whole 4-byte vector:

$a(x) = a_3x^3 + a_2x^2 + a_1x + a_0$
$b(x) = b_3x^3 + b_2x^2 + b_1x + b_0$

Then…

$$a(x)+b(x) = (a_3 \oplus b_3)x^3 + (a_2 \oplus b_2)x^2 + (a_1 \oplus b_1)x + (a_3 \oplus b_3). \qquad \textbf{(3.5)}$$

3.4.2 Multiplication

Multiplication using the new polynomials works in the same way as with the previous polynomial representation except for the fact that the coefficients are now elements of GF (2^8) and not GF (2). This means that any arithmetic operations carried out between the coefficients must be done using the rules set out above for elements in GF (2^8).

Once again a complication appears with the order of the resulting polynomial being too large to be represented as a 4-byte vector. Therefore the result must be reduced by calculating the result modulo *m(x)*, where *m(x)* is the irreducible polynomial

$$m(x) = x^4 + 1. \qquad \textbf{(3.6)}$$

The coefficients of the resulting polynomial $b(x) := c(x).a(x) \bmod m(x)$ can be calculated by finding the remainders after long division with *m(x).*

$$b(x) = b_3x^3 + b_2x^2 + b_1x + b_0 \text{ where}$$

$$\begin{aligned}
b_0 &= c_0.a_0 \oplus c_3.a_1 \oplus c_2.a_2 \oplus c_1.a_3\\
b_1 &= c_1.a_0 \oplus c_0.a_1 \oplus c_3.a_2 \oplus c_2.a_3\\
b_2 &= c_2.a_0 \oplus c_1.a_1 \oplus c_0.a_2 \oplus c_3.a_3\\
b_3 &= c_3.a_0 \oplus c_2.a_1 \oplus c_1.a_2 \oplus c_0.a_3
\end{aligned}$$

This can be expressed using the following matrix.

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} c_0 & c_3 & c_2 & c_1 \\ c_1 & c_0 & c_3 & c_2 \\ c_2 & c_1 & c_0 & c_3 \\ c_3 & c_2 & c_1 & c_0 \end{bmatrix} . \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \qquad \textbf{(3.7)}$$

Chapter 4

Application Objectives

Obviously the first main objective for both implementations is the successful use of the algorithm on the plaintext entered by the user. To check that this goal has been achieved it is necessary to acquire some precompiled test data where the algorithm has already been correctly applied to a given input.

Both implementations will need to be able to handle the three different key lengths (128, 192, 256 bits) and adapt the algorithm accordingly as set out in the official AES specification.

The user must be able to enter (as arguments to the program) the key size, a 128 bit plaintext and the key. The application must use these arguments to calculate the correct ciphertext and display it back to the user.

This dissertation is not concerned with the possible applications that may use the encryption technique, only the functionality of the actual encryption itself. Therefore there is no requirement to have an extensive user interface on either implementation. Instead the program only is only required to accept arguments through the command line to test the working of the algorithm against test data.

4.1 Implementation 1

This implementation is for use in standard home computers where the amount of memory available is large enough to make the amount of memory space available to the algorithm effectively unlimited.

The algorithm should use as many techniques as possible to increase the speed of execution at the cost of memory space.

4.2 Implementation 2

This implementation is for use in situations where memory space is at a premium. The algorithm should use a little memory space as possible while at the same time use as many techniques as possible to increase the speed without adding any extra memory usage.

4.3 Extensions

If time allows, there are further implementation ideas that can be considered.

These include creating hybrid implementations where techniques from both solutions can be combined to create a spectrum of solutions which could be used in a variety of different circumstances.

Another possible extension could be to implement the decryption algorithms for both implementations. Testing and analysis could then be carried out on the computational time taken by the decryption algorithms and comparisons can be made with the equivalent encryption algorithm to see whether or not the results agree.

Chapter 5

The Algorithm

This chapter explains the different parts of the AES algorithm and how they link together to create the cipher. The user will initiate the procedure by entering a 128 bit plaintext (32 hexadecimal characters) and a 128, 192, or 256 bit key.

For this chapter a number of symbols are used to shorten the notation:

Nb –	Refers to the number of words in the input plaintext (i.e. number of bits in the plaintext / 32)
Nk –	Number of words in the input key (i.e. number of bits in the key / 32)
Nr –	The number of rounds in the encryption algorithm.

Note that in the specification for Rijndael encryption, it allows for plaintext block sizes of 128, 192, 256 bits. However, in the AES specification this was altered to only allow 128 bit block sizes. This means that Nb (The number of words in the input plaintext) is always 128 bit / 32 = 4.

5.1 State and Key Data Structure

As mentioned earlier, the different operations of the algorithm act upon a matrix which holds the data being encrypted. This matrix is called the “state”. At the start of the encryption process the state holds the plaintext and by the end contains the cipher (encrypted) text. Each entry of the matrix holds a single byte of data.

The size of this state matrix is defined as [4 x Nb]. As explained above, in the AES specification, Nb is always equal to 4. This means that the state is a matrix containing 16 entries, each holding a byte of data (16 bytes = 128 bits).

The diagram below shows how the input is initially arranged in the state matrix:

PLAINTEXT: $p_0\, p_1\, p_2\, p_3 \ldots\ldots\ldots p_{15}$

INTITIAL STATE:

$$\begin{array}{cc} \begin{pmatrix} p_0 & p_4 & p_8 & p_{12} \\ p_1 & p_5 & p_9 & p_{13} \\ p_2 & p_6 & p_{10} & p_{14} \\ p_3 & p_7 & p_{11} & p_{15} \end{pmatrix} & \updownarrow \text{ 4 Rows} \\ \xleftrightarrow{\hspace{4em}} & \\ \text{Nb Cols} & \end{array}$$

The input key is also placed in a matrix in the same configuration as is shown above. However, the key is not restricted to just being 128 bit. To compensate for this the matrix which holds the key has extra columns if required. For example if we are working with a key length of 256 bits:

KEY : $k_0\, k_1\, k_2\, k_3 \ldots\ldots.. k_{31}$

MATRIX:

$$\begin{array}{cc} \begin{pmatrix} k_0 & k_4 & k_8 & k_{12} & k_{16} & k_{20} & k_{24} & k_{28} \\ k_1 & k_5 & k_9 & k_{13} & k_{17} & k_{21} & k_{25} & k_{29} \\ k_2 & k_5 & k_{10} & k_{14} & k_{18} & k_{22} & k_{26} & k_{30} \\ k_3 & k_7 & k_{11} & k_{15} & k_{19} & k_{23} & k_{27} & k_{31} \end{pmatrix} & \updownarrow \text{ 4 Rows} \\ \xleftrightarrow{\hspace{10em}} & \\ \text{Nk Cols} & \end{array}$$

5.2 The Rounds

The encryption algorithm is made up of a number of rounds. Each round contains a set routine of transformations which are applied in a certain order. There more rounds there are the harder the encryption becomes to break. In AES the rounds are made up of the following transformations:

1. **SubByte**
2. **ShiftRow**
3. **MixColumns**
4. **AddRoundKey**

The only exception to this rule is in the final time the round is applied. In this instance the MixColumns step is omitted. Each of the transforms above accept the state matrix as the input and produce an new updated state matrix.

5.2.1 Number of Rounds

The number of rounds that occur in the algorithm is dependant on the size of the key and the plaintext. The table below shows the number of rounds required depending on Nr and Nb as given in the Rijndael specification.

	Nb = 4	Nb = 6	Nb = 8
Nk = 4	10	12	14
Nk = 6	12	12	14
Nk = 8	14	14	14

Table 2: Number of Rounds depending on Nb and Nk

Since AES only allows 128 bit block sizes we only need to consider the column where Nb is equal to 4.

5.3 SubByte

The SubByte transformation is a substitution with acts upon each byte of the matrix individually. The transformation can be split into two separate operations.

The first of these operations is to find the multiplicative inverse of the byte as an element of GF (2^8) as described in chapter 3. As a rule, the byte 00_{16} is mapped onto itself since it does not have an inverse value. The process of calculating the inverse is non-linear and therefore makes the SubByte transformation non-linear as well.

Once the inverse has been found it is split into binary bits and an affine transformation is applied over the field GF(2). This transformation f is defined as:

b = f (a) *such that*

$$\begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \qquad \textbf{(5.1)}$$

where *b_i is the ith bit of byte b*
a_i is the ith bit of byte a

The complete SubByte transformation can be described as:

$$b = f(a^{-1}) \qquad \textbf{(5.2)}$$

5.4 ShiftRows

This transformation performs cyclic shifts of the state matrix rows. In Rijndael the offsets of the shift are dependant on the plaintext block size and the row it is to be applied to. Since in our AES implementations the block size is fixed, it is only dependant on the row number.

Row	Shift Ofset
0	0
1	1
2	2
3	3

Table 3: Cyclic shift offsets depending on row number

An example of the ShiftRow transformation is shown below:

$$\begin{bmatrix} 00 & 11 & 22 & 33 \\ 44 & 55 & 66 & 77 \\ 88 & 99 & AA & BB \\ CC & DD & EE & FF \end{bmatrix} \longrightarrow \text{ShiftRow} \longrightarrow \begin{bmatrix} 00 & 11 & 22 & 33 \\ 55 & 66 & 77 & 44 \\ AA & BB & 88 & 99 \\ FF & CC & DD & EE \end{bmatrix}$$

Figure 2: Example of ShiftRow Transformation

5.5 MixColumns

This step operates on whole columns of the state matrix. The columns are represented as polynomials with coefficients in GF (2^8) as discussed in chapter 3. This polynomial is multiplied modulo *m(x)* with a set polynomial *c(x)* and the resulting polynomial replaces the column.

In the specification the set polynomials are given as:

$$m(x) = x^4 + 1$$
$$c(x) = 03.x^3 + 01.x^2 + 01.x + 02$$

Therefore the MixColumns transformation can be summarised by:

$$b(x) = a(x) . (03.x^3 + 01.x^2 + 01.x + 02) \bmod (x^4+1). \qquad \textbf{(5.3)}$$

As discussed in (3.4.2), it is possible to represent multiplication of polynomials with coefficients in GF (2^8) in terms of matrices by expanding the terms of the multiplication and applying long division by (x^4+1). Substituting *c(x)* into equation ***(3.7)*** we get the following representation of the MixColumns transform.

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Figure 3: MixColumns transform represented by matrices

Note that when the matrix multiplication takes place, the operands will be bytes and therefore must be multiplied using the set rules for multiplication in GF (2^8).

5.6 AddRoundKey

This is the final transformation of the round, and conceivably the most straightforward.

Once the key is placed into a matrix it is expanded using a function called KeyExpansion (see below) to create Nr + 1 different [4 x 4] matrices called round keys. At the start of the algorithm the first round key matrix is introduced into the state by means of a bitwise XOR of corresponding bytes. Then round by round the remaining Nr round keys are introduced by the same method.

$$\begin{bmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{bmatrix} \oplus \begin{bmatrix} k_0 & k_4 & k_8 & k_{12} \\ k_1 & k_5 & k_9 & k_{13} \\ k_2 & k_6 & k_{10} & k_{14} \\ k_3 & k_7 & k_{11} & k_{15} \end{bmatrix} = \begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix}$$

Figure 4: The expanded key matrix is combined with the state by a bitwise XOR

5.7 Key Schedule

The AddRoundKey function above requires (Nr + 1) round keys which are created using an expanding function acting on the input key. This expanded key is held in an array of 4*(Nr + 1) words and is accessed every time an AddRoundKey transform is required. This array is filled recursively using previous words starting with the original key.

Obviously the size of this array is dependant on the size of the key since Nr is derived from Nk (the number of words in the key). In addition to this the recursive algorithm differs depending on Nk.

For Nk ≤ 6:

```
KeyExpansion(byte Key[4*Nk] word W[Nb*(Nr+1)])
{
   for(i = 0; i < Nk; i++)
        W[i] = (Key[4*i],Key[4*i+1],Key[4*i+2],Key[4*i+3]);
   for(i = Nk; i < Nb * (Nr + 1); i++)
   {
        temp = W[i - 1];
        if (i % Nk == 0)
           temp = SubByte(RotByte(temp)) ^ Rcon[i / Nk];
        W[i] = W[i - Nk] ^ temp;
   }
}
```

And for Nk > 6:

```
KeyExpansion(byte Key[4*Nk] word W[Nb*(Nr+1)])
{
   for(i = 0; i < Nk; i++)
      W[i] = (key[4*i],key[4*i+1],key[4*i+2],key[4*i+3]);
   for(i = Nk; i < Nb * (Nr + 1); i++)
   {
      temp = W[i - 1];
      if (i % Nk == 0)
         temp = SubByte(RotByte(temp)) ^ Rcon[i / Nk];
      else if (i % Nk == 4)
         temp = SubByte(temp);
      W[i] = W[i - Nk] ^ temp;
   }
}
```

Pseudocode taken from (Daemen and Rijmen 1999)

In the above algorithms:

Key[]	is the array holding the input key.
W[]	is the array holding the expanded key.
SubByte	refers to same transformation as in (5.3).
RotByte()	is a function that returns a word where the bytes have performed a cyclic rotation.
Rcon[i]	is the row vector containing the bytes: [02^{i-1}, 00, 00, 00].

It is clear to see that the first loop of the algorithm copys the input key into the first Nk words of the expanded key. The second loop defines the remaining words using the (i-1)th word and the (i-Nk)th word with various transformations applied to them at certain intervals.

5.8 The Cipher Structure

Using the defined transformations and key expansion from above it is possible to define the full cipher:

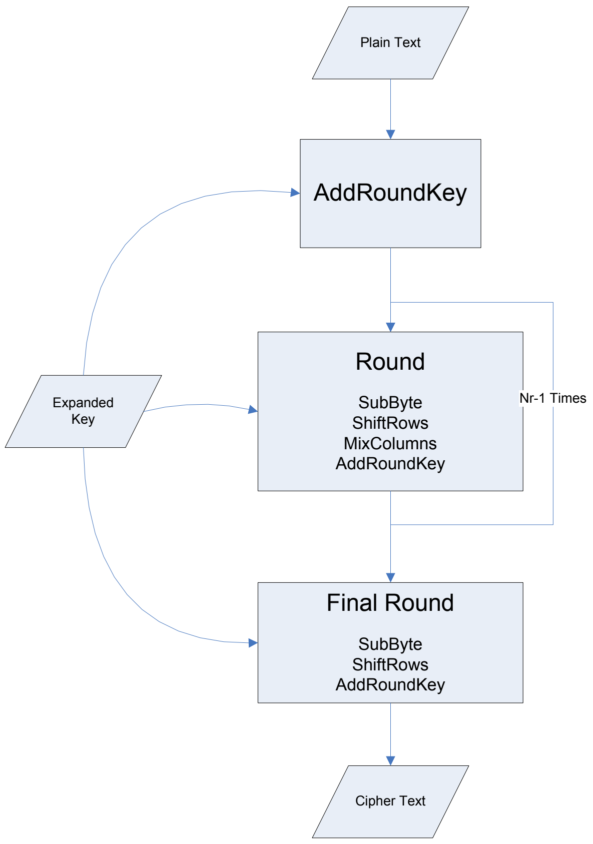


Figure 5: The structure of the AES cipher

This algorithm has been designed in such a way that each bit of the state is dependant on all bits of the state two rounds earlier. This high amount of diffusion makes the encryption extremely strong.

Chapter 6

Implementation Issues

6.1 Polynomial Multiplication

As it has been shown, in order to carry out multiplication of elements of GF (2^8) it is necessary to implement a way to calculate multiplications of polynomials.

6.1.1 Multiplication By Shifts

There are a number of different methods which can be used to carry out this operation. One of these methods is the same as pupils are taught to do long multiplication at school. However, to use this exact method in the algorithm would be inefficient. For this reason it is necessary to find optimisations to enhance this technique.

If we take the polynomial x which represents the byte 02_{16} it can be seen that the multiplication of this polynomial amounts to a single left shift of the bits of the byte being multiplied. Similarly, multiplication by the polynomial x^2 results in the bits being shifted twice to the left. This technique also works for higher powers of the variable x. Using this knowledge it is possible to adapt the original long multiplication method to create an algorithm which works very quickly at the CPU level. Brian Gladman (Gladman 2003) explains the way it is possible to use a combination of shifting (to produce the multiplications of the powers of x) and XOR operations (to add them together) to produce the required result.

This method only requires a small amount of memory whilst keeping the speed of the operation high. This makes it very useful for the low-memory implementation.

6.1.2 Multiplication Using Log Tables

There exists a method which requires more memory but produces faster results. This method uses logarithm tables to calculate multiplications very quickly. In order to create logarithm tables for GF (2^8) a "field generator" is required. A generator is an element of the field which when multiplied by itself a finite number of times can produce every element of that field. In GF (2^8) one such generator is the byte 03_{16}. This means that if you keep multiplying 03_{16} by itself it will eventually equal every element of GF (2^8). Therefore using the following algorithm we can produce log and anti-log tables:

```
temp = 1;

  for(i=0; i<256; i++){

    LogTable[temp] = i;
    ALogTable[i] = temp;
    temp  = Poly_Multiplication(temp,3);

  }
```

Once these two tables have been initialised it is possible to calculate the multiplication of two bytes (a and b) using the following statement:

```
ALogTable[(LogTable[a]+LogTable[b])%255];
```

Therefore to multiply any two bytes using this method we only need two array lookups and one addition (and possibly a subtraction). However, even though this method is very quick, the log tables need to be initialised first to use it. This requires 255 polynomial multiplications using the method described earlier. The advantage is that once this is completed the tables can be reused throughout the whole of the encryption process. For this reason this method will be incorporated into the high speed implementation.

6.2 SubByte Transformation

6.2.1 Multiplicative Inverses

As mentioned in the previous chapter the first operation needed in order to complete a SubByte transformation is finding the multiplicative inverse of a given byte. Also explained was the fact that this can be done using the Extended Euclidean Algorithm.

The original Euclidean Algorithm was used to find the greatest common divisor between two integers:

```
function gcd(a, b)
     if b = 0
         return a
     else
         return gcd(b, a modulus b);
```

Soon after this was first introduced an extension of the algorithm called the Binary Euclidean Algorithm was produced. This algorithm took advantage of the binary structure of the integers which allowed for divisions to be replaced by bitwise operations and therefore increase efficiency.

Unfortunately, to find the inverse of a byte, we are uninterested in the greatest common divisor itself but instead, we require the values *x* and *y* such that:

$$ax + by = gcd(a, b) \qquad \textbf{(6.1)}$$

The Extended Euclidean Algorithm does exactly this allowing us to calculate the inverse of a byte as demonstrated in equation **(3.4)**. This version of the algorithm can also use the structure of binary numbers to increase efficiency and will be used in the new implementations.

6.2.2 Affine Transformation

The second of the operations is the linear transform described by the matrix function:

$$\begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

Figure 6: Matrix Representation of the Affine Transform

This function can be summarised by the following:

$$b_i = a_i \oplus a_{(i+4)\bmod 8} \oplus a_{(i+5)\bmod 8} \oplus a_{(i+6)\bmod 8} \oplus a_{(i+7)\bmod 8} \oplus c_i \qquad \textbf{(6.2)}$$

where a_l is the ith bit of the byte being transformed
b_i is the ith bit of the resulting byte
c_i is the ith bit of the byte 63_{16}

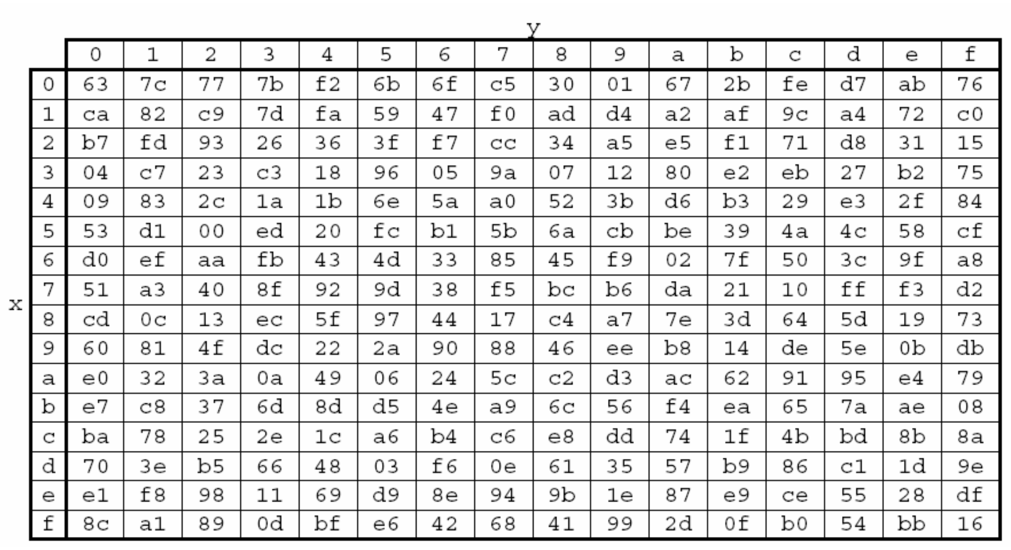
In order to use this new equation in the implementations it is necessary to create a short function for returning the ith bit of a byte. This is synonymous to returning the result of the following inequality:

$$a \bmod (2^{i+1}) \geq 2^i \qquad \textbf{(6.3)}$$

where *a* is the byte
i is the bit position

6.2.3 Optimisation

It is possible to speed up the operation of calculating a SubByte transform through the use of a lookup table (usually called an s-box). This table holds the result of finding the inverse and applying the affine transformation to each byte which can then be used when ever a SubByte transform is required for a byte.



x \ y	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

Table 4: AES S-box for byte XY_{16}

This lookup table will be much quicker than calculating on-the-fly since each transform only requires one array lookup. To initialise the table it will require finding the inverse and applying the affine transfer to all 256 possible bytes. Once this is complete it can be used during the remainder of the encryption process extremely quickly.

This table requires 256 bytes of memory space and therefore will not be suitable for the low memory version since calculating the SubByte routine on-the-fly uses only a tiny fraction of this memory allocation. However this table can be implemented in the high speed implementation as the memory usage is of little interest.

6.3 ShiftRow Transformation

To reduce the amount of memory usage the ShiftRow transformation can carry out the process row by row. This means calculating the result of the transformation on one single row and then replacing it in the state matrix before working on the remaining rows. This method requires a temporary 4-byte array to hold the shifted rows until it is copied back into the state matrix.

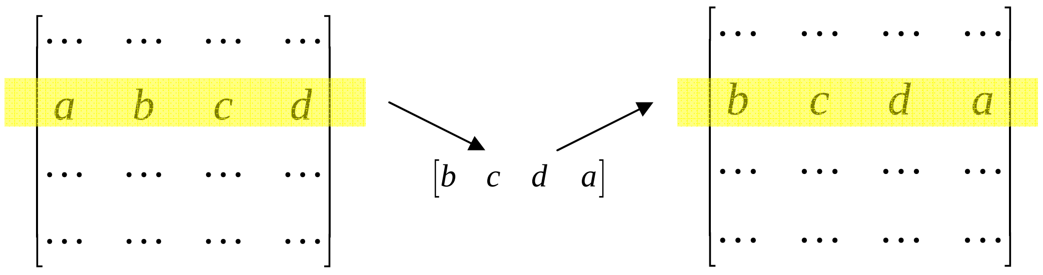


Figure 7: Diagram showing the working of ShiftRow using temporary row

The alternative to this is to calculate the result of the transformation for the whole matrix using a temporary [4x4] array of bytes, then overwrite the whole state matrix with the temporary one in a single pass. Since this method uses more memory it should only be used in the high speed implementation whereas the low-memory solution must use the first method mentioned.

6.4 MixColumns Transformation

As discovered in the previous chapter this transformation can also be represented by a matrix multiplication.

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Figure 8: MixColumns transform represented by matrix multiplication

To calculate this matrix multiplication it is necessary to conduct multiplication of elements in GF (2^8). The method for this calculation then depends on which implementation it is carried out on (see 6.1).

In this transformation however, the multiplication carried out will always include one of the bytes 01_{16}, 02_{16} or 03_{16}. These particular bytes are easy to implement since multiplying by 01_{16} does not alter a byte, multiplying by 02_{16} refers to a single left shift of the bits, and multiplying by 03_{16} (represented as $x+1$) can be created from the combination of a left shift and an XOR with the original byte.

Using this knowledge it is fairly straightforward to create an algorithm that calculates the MixColumn transformation which uses fast bitwise operations and does not draw on temporary memory space.

6.5 Key Schedule

6.5.1 Key Expansion

The array holding the expanded key needs to be accessed each round by the AddRoundKey function. To make this process as simple as possible the array can be structured so that it is easy to identify sections of the array which refer to specific rounds. To achieve this, the array should be a collection of (Nr-1) [4x4] arrays which can be mapped directly on top of the state matrix with an XOR operation.

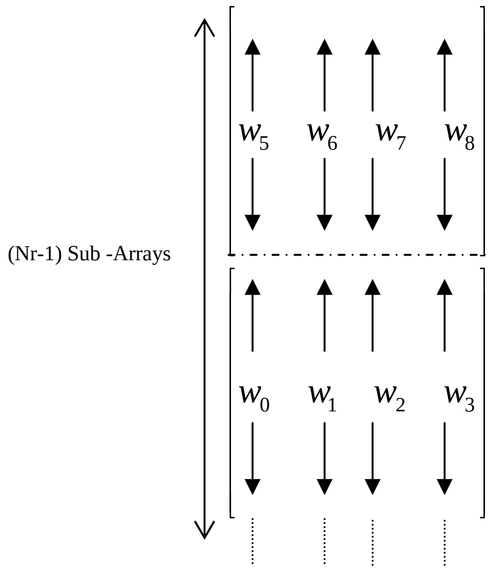


Figure 9: Structure of the Expanded Key Array

In the above diagram w_i refers to the i^{th} word created using the recursive key expansion algorithm.

Once the array is arranged in this way it is easy to find the address of the top left element of any particular round key using the following formula:

RoundKey_TopLeft = ExpandedKeyArray[4 x RoundNumber][0]

Note that the key referring to round one starts at row 4. This is because the cipher commences with an AddRoundKey before the first round and it is here where the first 4 rows of the extended key are introduced into the state matrix with an XOR operation.

6.5.2 Initialising Functions

The final operation that needs to be considered is the row vector:

$$Rcon[i] = [\ 02^{i-1}, 00, 00, 00\ ]$$

This array is required by the AddRoundKey transformation and the value of *i* refers to the current round number. The different values of 02^{i-1} can be initially calculated one after another by constantly multiplying by 02_{16} and the saving each result into an array. This is quicker than having to calculate the result of 02^{i-1} independently each time the AddRoundKey requires it. Once again this will require extra memory space and therefore will only be suitable for the high speed implementation. The low memory solution will have to calculate the values each round on-the-fly.

Chapter 7

Formal Descriptions

7.1 Implementation 1 – High Speed

7.1.1 Initialising Functions

Name:	**Rcon_Initialise**
Input:	Empty 16-byte Rcon Array
Output:	Rcon Array
Action:	Fill the empty array with values of 02^{i-1}

Name:	**SBox_Initialise**
Input:	Empty 256-byte S-Box Array
Output:	S-Box Array
Action:	Fill the empty array with substitution values used by SubByte using AffTransform and ExtendEuclid functions

Name:	**LogTable_Initialise**
Input:	Empty 2 x 256-byte Arrays
Output	Arrays
Action:	Fill array with logarithm and anti-logarithm values using the field generator 03_{16} and PolynomialMult function.

Name:	**GetPlainText**
Input:	128-bit Hexadecimal String, [4x4]-byte State Array
Output:	State Array
Action:	Convert string to bytes and place into state array in order

Name:	**GetKeyText**
Input:	128, 192 or 256-bit Hexadecimal String, [4xNk]-byte Key Array
Output:	Key Array
Action:	Convert string into bytes and place into key array in order.

Name:	**KeyExpansion**
Input:	[4xNk]-byte Key Array, [(4x(Nr+1))x4]-byte ExpandedKey Array
Output:	ExpandedKey Array
Action:	Apply the Key Expansion routine on Key Array to produce ExpandedKey Array.

7.1.2 Transformation Functions

Name:	**SubByte**
Input:	[4x4]-byte State Array
Output:	State Array
Action:	Replace each byte in array using S-box Array

Name:	**MixColumns**
Input:	[4x4]-byte State Array
Output	State Array
Action:	Apply the MixColumn transform to the State Array

Name:	**ShiftRow**
Input:	[4x4]-byte State Array
Output:	State Array
Action:	Apply the ShiftRow transform on the State Array

Name:	**AddRoundKey**
Input:	[4x4]-byte State Array, [(4x(Nr+1))x4]-byte ExpandedKey Array
Output:	State Array
Action:	XOR values of Round Sub-Array of ExpandedKey with State Array.

7.1.3 Other Functions

Name:	**PolynomialMult**
Input:	Byte A, Byte B
Output:	Byte C
Action:	Returns C:= A multiply B using GF(2^8) representation

---

Name:	**PrintState**
Input:	[4x4]-byte State Array
Output:	-

Action:	Prints the State Array as an output string

Name:	**ExtendedEuclidean**
Input:	Byte
Output:	Byte
Actions:	Returns to multiplicative inverse of the input byte

Name:	**AffineTransform**
Input:	Byte
Output:	Byte
Action:	Returns the result of the Affine Transformation applied to the input byte.

Name:	**ByteBit**
Input:	Byte, Integer *i*
Output:	1 or 0
Action:	Returns the value of the i^{th} bit of the byte

Name:	**LogMultiplication**
Input:	Byte A, Byte B
Output:	Byte C
Action:	returns the result of C:= A multiply B using log/anti-log tables.

7.2 Implementation 2 – Low Memory

7.2.1 Initialising Functions

Name:	**GetPlainText**
Input:	128-bit Hexadecimal String, [4x4]-byte State Array
Output:	State Array
Action:	Convert string to bytes and place into state array in order

---

Name:	**GetKeyText**
Input:	128, 192 or 256-bit Hexadecimal String, [4xNk]-byte Key Array
Output:	Key Array
Action:	Convert string into bytes and place into key array in order.

Name:	**KeyExpansion**

Input:	[4xNk]-byte Key Array, [(4x(Nr+1))x4]-byte ExpandedKey Array
Output:	ExpandedKey Array
Action:	Apply the Key Expansion routine on Key Array to produce ExpandedKey Array.

7.2.2 Transformation Functions

Name:	**SubByte**
Input:	[4x4]-byte State Array
Output:	State Array
Action:	Replace each byte in array using S-box Array

Name:	**MixColumns**
Input:	[4x4]-byte State Array
Output	State Array
Action:	Apply the MixColumns transform to the State Array

Name:	**ShiftRow**
Input:	[4x4]-byte State Array
Output:	State Array
Action:	Apply the ShiftRow transform on the State Array

Name:	**AddRoundKey**
Input:	[4x4]-byte State Array, [(4x(Nr+1))x4]-byte ExpandedKey Array
Output:	State Array
Action:	XOR values of Round Sub-Array of ExpandedKey with State Array.

7.2.3 Other Functions

Name:	**PolynomialMult**
Input:	Byte A, Byte B
Output:	Byte C
Action:	Returns C:= A multiply B using GF(2^8) representation

---

Name:	**PrintState**
Input:	[4x4]-byte State Array
Output:	-
Action:	Prints the State Array as an output string

Name:	**ExtendedEuclidean**
Input:	Byte
Output:	Byte
Actions:	Returns to multiplicative inverse of the input byte

Name:	**AffineTransform**
Input:	Byte
Output:	Byte
Action:	Returns the result of the Affine Transfromation applied to the input byte.

Name:	**ByteBit**
Input:	Byte, Integer *i*
Output:	1 or 0
Action:	Returns the value of the i^{th} bit of the byte

Name:	**CopyRow**
Input:	4-byte Temp Array, [4x4]-byte State Array, integer *i*
Output:	State Array
Action:	Copy across contents of Temp Array into row *i* of State Array

Chapter 8

Function Walkthroughs (Low-Level)

This chapter aims to provide the reader with an understanding of exactly how the encryption algorithm is implemented. The first section provides a walkthrough of the functions which are common to both implementations. The next two sections refer to functions which are specifically written for just one of the implementations.

8.1 Common Functions

8.1.1 Multiplication in GF(2^8) (Using Shifts)

This function carries out the multiplication of two bytes.

```
byte GFPolyMult( byte a, byte b)
{
  byte temp;
  int r=0;
```

The function loops until all shifts have been completed

```
  while (a != 0)
  {
```

If the least significant bit of the first byte is a one then we need to add (XOR) the other byte to the running total.

```
    if ((a & 1) != 0)
      r = r ^ b;
```

The second byte must now be shifted to the left but if this is going to create a number larger than 256 then m(x) must be subtracted (XOR operation)

```
    temp = b & 0x80;
    b = b << 1;
    if (temp != 0)
      b = b ^ 0x1b;
```

The first byte must now be shifted the right before the process is repeated.

```
    a = a >> 1;
```

```
  }
```

Finally, once all the bits of the first byte have been dealt with the running total is returned.

```
  return r;
}
```

8.1.2 Retrieving the Plaintext

This function converts the input plaintext into hexadecimal values and places them into the state.

```
getPlainText (byte array[4][4], char *text)
{
  long value;
  byte temp[1];
  byte * end;
  int i,j;
```

Each entry of the state is dealt with in order

```
  for (i=0; i<4; i++){
    for (j=0; j<4; j++){
```

Now the two characters relating to the specific state entry are placed in a temporary array

```
      temp[0] = text[(8*i)+(2*j)];
      temp[1] = text[(8*i)+(2*j)+1];
```

The temporary array now holds a string of two characters which is converted into hexadecimal values using `strtol()`.

```
      array[j][i] =(int) strtol(temp, &end, 16);
    }
  }

}
```

8.1.3 Retrieving the Key

This function is extremely similar to the previous function except for the fact that it must handle keys of different sizes.

```
getKeyText (byte array[4][8], char *text)
{
  long value;
  byte temp[1];
  byte * end;
  int i,j;
```

The width of the key array can be different hence the first loop statement runs from column 0 to column Nk-1. The remainder of the code is the same as with the plaintext retrieval

```
  for (i=0; i<Nk; i++){
    for (j=0; j<4; j++){
      temp[0] = text[(8*i)+(2*j)];
      temp[1] = text[(8*i)+(2*j)+1];
      array[j][i] =(int) strtol(temp, &end, 16);
    }
  }

}
```

8.1.4 Printing the State to Screen

This function is used to print (to the screen) the state array as a string.

```
printState( byte array[4][4])
{
  int i,j;
```

Each element of the state array is printed to the screen in turn using a loop.

```
  for (i=0; i<4; i++)
  {
    for (j=0; j<4; j++)
    {
```

The values are formatted in uppercase hexadecimal characters.

```
      printf("%02X", array[j][i]);
    }

  }printf("\n");
}
```

8.1.5 Mix Columns Transformation

The below code has been designed using the optimisations explained in chapter 6.

```
mixColumns (byte state[4][4])
{
  byte ab, cd, abcd;
  int j;
```

The function loops through the four columns independently.

```
  for(j=0; j<4; j++){
```

Three XOR operations are initially made to make the calculations below a bit more efficient.

```
    ab = state[0][j] ^ state[1][j];
    cd = state[2][j] ^ state[3][j];
    abcd = ab ^ cd;
```

Each entry of the row is calculated using the matrix form of the MixColumns transformation. (see 6.4).

```
    state[3][j] = state[3][j] ^ abcd ^
                  LogMult(2,(state[3][j]^state[0][j]));
    state[1][j] = state[1][j] ^ abcd ^
                  LogMult(2,(state[1][j]^state[2][j]));
    state[0][j] = state[0][j] ^ abcd ^ LogMult(2,ab);
    state[2][j] = state[2][j] ^ abcd ^ LogMult(2,cd);

  }
}
```

8.1.6 Extended Euclidean Algorithm

This algorithm returns the inverse of a given byte.

```
byte extEuclid (byte a)
{
  int m;
  byte b,d;
```

An exception case is needed if the input byte is zero. The inverse of this byte is zero by definition.

```
  if (a==0){
    return 0;
  }
```

The variables used throughout the algorithm are to be initialised

```
  m = 0x11b;
  b = 0; d = 1;
```

The algorithm must loop until the remainder variable reaches 0 which indicates that the greatest common divisor has been found

```
  while (m!=0){
```

Whilst the remainder variable is a multiple of 2 we can keep reducing it with shifts.

```
    while ((m%2)==0){

      m = m >> 1;
      if ((b%2)==0){
        b = b >> 1;
      }else{
        b = ((b^0x11b)>>1);
      }

    }
```

Whilst a is a multiple of 2 we can keep reducing it with shifts also.

```
    while ((a%2)==0){

      a = a >> 1;

      if ((d%2)==0){
        d = d >> 1;
      }
      else{
        d = ((d^0x11b)>>1);
      }

    }
```

We now determine the variables for the next loop depending on whether the variable a is bigger than the remainder variable m.

```
    if (m>=a){
      m = m ^ a;
      b = b ^ d;
    }
    else{
      a = a ^ m;
      d = d ^ b;
    }

  }
```

Once the greatest common divisor has been found we know what the inverse of the original byte is. The function returns this inverse byte.

```
  return d;
}
```

8.1.8 Byte Bit Function

This routine returns the value of the bit in the ith position of the byte. It is used in the affine transform algorithm.

```
int byteBit(int number, int bit)
{
```

The value of the ith bit is the same as the result of $a \bmod (2^{i+1}) \geq 2^i$:

```
  return (number%(pow(2,bit+1)))>=pow(2,bit);
}
```

8.1.9 Affine Transformation

This is the second part of the SubByte routine and is expressed by the following function.

```
byte affTrans(byte a)
{
  int i;
  byte result, temp;
```

A variable is required to total the value of the byte from the bits calculated

```
  result = 0;
```

Each bit of the new byte is calculated independently and then it’s respective value added to the total.

```
  for (i=0; i<8; i++){
```

This calculation below is equivalent to equation **(6.2)**

```
    temp = byteBit(a,i) ^ byteBit(a,(i+4)%8) ^
             byteBit(a,(i+5)%8) ^ byteBit(a,(i+6)%8) ^
             byteBit(a,(i+7)%8);
    result = result + (byte)(pow(2,i)*temp);
  }
```

The last part of the transform is to XOR a constant byte. The final result is then returned.

```
  return (result ^ 0x63);
}
```

8.1.10 Add Round Key

The final function which is common to both implementations is the AddRoundKey transform. This introduces the round key into the state by means of a bitwise XOR operation. This process is carried out byte-by-byte.

```
addRoundKey(byte state[4][4], int round, byte ExpKey[44][4])
{
  int i,j;

  for (i=0; i<4; i++){
    for (j=0; j<4; j++){
```

The relevant sub-array of the expanded key must by calculated as shown below in the array index.

```
      state[i][j] = state[i][j] ^ ExpKey[(round)*4 + i][j];
    }
  }
}
```

8.2 High Speed Specific Functions

8.2.1 Key Expansion

This routine accepts the key array and fills an empty expanded key array with the round keys used by the addRoundKey function below.

```
keyExpansion(byte key[4][8], byte W[][4])
{
  int i,j,a,b,c,d;
  byte temp[4];
  byte tem;
```

Throughout the function a and b are used as pointers to the top of the current word being calculated.

```
  a=0; b=0;
```

The routine starts by moving the first Nk words of data into the first Nk words of the expanded key array called W.

```
  for (j=0; j<Nk; j++){

    for(i=0; i<4; i++){
      W[a+i][b] = key[i][j];
    }
```

We move the values of a and b along to the next word.

```
    b++;
    if (b==4){
      b=0; a=a+4;
    }
  }
```

The remainder of the expanded key is calculated recursively from previous words.

```
  for (i=Nk; i<((Nr+1)*4); i++){
```

The previous word is copied into a temporary array. Variables d and c refer to the start of the previous word and are calculated from a and b.

```
    if (b==0){
      d=3;
      c = a-4;
    }else{
      d = b-1;
      c = a;
    }
    for (j=0; j<4; j++){
      temp[j] = W[c+j][d];
    }
```

Additional transformations are done to this temporary array if the current word number is a multiple of Nk.

```
    if (i%Nk == 0){
```

Firstly the values of the array are replaced using the SubByte transformation. In this high speed version it is done using the s-box.

```
      for (j=0; j<4; j++){
         temp[j] = SBox[temp[j]];
      }
```

Secondly a cyclic shift is performed on the temporary array.

```
      tem = temp[0];
      for (j=0; j<3; j++){
         temp[j]=temp[j+1];
      }
      temp[3] = tem;
```

Finally the first byte of the temporary word is XOR'd with the relevant entry of the Rcon lookup table (different from low memory version).

```
      temp[0] = temp[0] ^ Rcon[(i/Nk)-1];
    }else if((Nk > 6) && (i%Nk==4)){
```

If Nk is larger than 6 then there is an added transform for the temporary array.

```
      for (j=0; j<4; j++){
         temp[j] = SBox[temp[j]];
      }
    }
```

The temporary array is XOR'd with the word 4 positions earlier, and is placed in the expanded key matrix as the new word. Variables c and d are used again to find the word 4 places earlier.

```
    c=a-4; d=b;
    for(j=0; j<(Nk-4); j++){
      if(d==0){
         d=3;
         c=c-4;
      }else{
         d--;
      }
    }
    for (j=0; j<4; j++){
      W[a+j][b] = temp[j] ^ W[c+j][d];
    }
```

The pointers a and b are moved on to the next word in the array and the process is repeated.

```
    b++;
    if (b==4)
      b=0; a=a+4;
  }
}
```

8.2.2 Lookup Table Initialisation

The initialisation process is the most important routine for utilising the available memory space and speeding up the cipher. The function fills the empty lookup tables using the standard functions which would otherwise be used to calculate the transformations on-the-fly.

```
initialise()
{
  int i,temp;
```

Firstly the various values of $(02_{16})^i$ are placed into the Rcon lookup table.

```
  temp = 0x01;
  Rcon[0] = temp;
```

The powers must be calculated using the standard GF(2^8) representation.

```
  for (i=1; i<15; i++){
    temp=GFPolyMult(temp, 2);
    Rcon[i] = temp;
  }
```

The S-box is then filled with the SubByte transformation of all 256 possible bytes.

```
  for(i=0; i<256; i++){
    SBox[i]=affTrans(extEuclid(i));
  }
```

Finally logarithm and anti-logarithm tables are initialised using the byte 03_{16} as the field generator.

```
  temp = 1;
  for(i=0; i<256; i++){
    LogTable[temp] = i;
    ALogTable[i] = temp;
    temp  = GFPolyMult(temp,3);
  }}
```

8.2.3 Array Copying

The ShiftRow transformation requires this function to copy an entire array to the state array. The algorithm is very simple; it takes two arrays as inputs and copies the values from one to another.

```
copyArray( byte from[4][4], byte to[4][4])
{
  int i,j;
  for (i=0; i<4; i++){
    for (j=0; j<4; j++){
      to[i][j] = from[i][j];
}}}
```

8.2.4 ShiftRow Transformation

This function computes the ShiftRow transformation on the state matrix. In this implementation a temporary array is used to hold the shifted rows. Once fully filled, the temporary array is copied over to the state array in one complete pass. This is in comparison to the low memory version where each row is shifted and then copied separately to save memory.

```
shiftRows( byte state[4][4])
{
  byte temp[4][4];
  int i,j,newcol;
```

The function loops over every row and every entry in that row.

```
  for (i=0; i<4; i++){
    for (j=0; j<4; j++){
```

For each element the new position in the row is calculated.

```
      newcol = (j+(4-i))%4;
      temp[i][newcol] = state[i][j];
    }
  }
  copyArray(temp, state);
}
```

8.2.5 SubByte Transformation

Since there are now lookup tables which can be used for this transformation there is no need to calculate the substitution on the fly. This function utilises the Sbox array created by the initialisation process to carry out the operation.

```
subByte(byte state[4][4])
{
  int i,j;
  for (i=0; i<4; i++){
    for (j=0; j<4; j++){
```

The value in the state is just replaced by its corresponding value found in the SBox array.

```
      state[i][j] = SBox[state[i][j]];
}}}
```

8.2.6 Multiplication in GF(2^8) (Log Tables)

This is another operation which can now be dealt with using lookup tables. The function returns the multiplication of the two bytes passed to it.

```
byte LogMult(byte a, byte b)
{
```

If either of the bytes being multiplied is equal to zero then the result is always zero.

```
  if (a==0 || b==0)
    return 0;
```

The basic laws of logarithms allow the operation to only require a simple arithmetic addition (mod 255).

```
  return ALogTable[(LogTable[a]+LogTable[b])%255];

}
```

8.2.7 "Main" Function

As the implementations are written in C the "main" function is the routine which actually runs the cipher as a whole. The only difference between this function and the one in the low memory implementation is the initialisation function call at the start.

This function also has another important role. It accepts the input strings entered into the command line and checks them for obvious user input errors.

```
main(int argc, char *argv[])
{
  int i,j,k,temp;
  byte W[60][4];
  byte Key[4][8];
  byte state[4][4];
```

If the user has not given the required 3 arguments in the command line then syntax help is printed to the screen.

```
  if (argc != 4){
    printf("syntax:\tAES [key size in bits] [plaintext (32 Hex
            Chars)] [key]\n");
    exit(0);
  }
```

The second command line argument is turned into a value and checked that it is one of the allowable key lengths.

```
  sscanf(argv[1],"%d", &temp);

  if (temp!=128 && temp!=192 && temp!= 256){
    printf("Invalid Key Size.  Must be 128, 192 or 256. Type
            AES for syntax help\n");
    exit(1);
  }
```

The final check makes sure that the plaintext length is correct and that the key length is consistent with the stated value.

```
  if (strlen(argv[2])!=32 || strlen(argv[3])!= (temp/4)){
    printf("Invalid length of plaintext or key, Type AES for
            syntax help\n");
    exit(1);
  }
```

Global values used throughout the code are initialised.

```
  sscanf(argv[1],"%d", &Nk);
  Nk = Nk/32;
  Nr = NoRounds[(Nk/2)-2];
```

The key and plaintext is retrieved and then the key is expanded.

```
  getKeyText(Key, argv[3]);
  keyExpansion(Key, W);
  getPlainText(state, argv[2]);
```

The initial AddRoundKey transformation is applied.

```
  addRoundKey(state,0,W);
```

The standard and final rounds are then applied to the state.

```
  for (i=1; i<Nr; i++){

    subByte(state);
    shiftRows(state);
    mixColumns(state);
    addRoundKey(state,i,W);
  }
  subByte(state);
  shiftRows(state);
  addRoundKey(state,Nr,W);
```

Finally the resulting ciphertext is printed to the screen.

```
  printState(state);
}
```

8.3 Low Memory Specific Functions

8.3.1 ShiftRow Transformation

This function differs from the other implementation since each row is transformed and copied separately reducing the need for a full state-sized temporary array.

```
shiftRows( byte state[4][4])
{
  byte temp[4][4];
  int i,j,newcol;
  for (i=0; i<4; i++)
  {
    for (j=0; j<4; j++)
    {
      newcol = (j+(4-i))%4;
      temp[i][newcol] = state[i][j];
    }
```

It is at this point where the temporary array is copied over, freeing it for use with the remaining rows. The function to do this is very similar to copyArray.

```
  copyRow(temp, state, i)
  }
}
```

8.3.2 Key Expansion

This function is very similar to the other implementations equivalent function and therefore there is no need to explain all the code again. The only difference between the two functions is that the low memory implementation carries out all transformations on the fly instead of using lookup tables such as SBox, Rcon and the logarithm tables.

8.3.3 “Main” Function

Once again there is no need to re-explain this function since it is fundamentally the same as with the high speed implementation. This time the only difference is that there is no need to call the lookup table initialisation function required by the high speed version.

Chapter 9

Implementation

The program names for the two implementations are as follows:

AES	-	Low Memory AES Implementation
QAES	-	For the Quick AES Implementation

These applications have been written in the C programming language and the source code for these two applications (AES.c and QAES.c) is available in the appendix.

The syntax to run the application is as follows:

(Q)AES [number of bits in key] [plaintext input] [key input]

The application will then return the ciphertext to the screen. For example:

```
amos$ AES 128 00112233445566778899AABBCCDDEEFF00112233445566778899AABBCCDDEEFF
62F679BE2BF0D931641E039CA3401BB2
amos$
```

Figure 10: A test run of the program 'AES' executed on the computer 'amos'

Chapter 10

Software Testing

This chapter deals with two types of testing of the written applications. The first section tests the accuracy of the encryption algorithm against precompiled test data to check that the correct output ciphertexts are being produced. The second section deals with the functionality of the program under incorrect inputs from the user.

10.1 Encryption Accuracy

In order to carry out this testing a set of precompiled test data is required to compare the results from the program against expected results. Such data is available from the National Institute of Standards and Technology’s (NIST) website (http://csrc.nist.gov/CryptoToolkit/aes/rijndael/) and has been compiled by the writers of the algorithm.

Each implementation is tested with 150 test values, 50 for each of the different key sizes. The output of the program is compared to the expected results given in the test values and the implementation is said to be a success if and only if all values compare correctly.

In order to process this number of encryption tests a shell script has been devised which asks the user which version of the program to test and the file where the test values can be found. It then carries out the operation and directs the output into a file. This shell script can be found in appendix C.

Tables showing the input, actual output, expected output and comparison results are also available in appendix C.

Fortunately the results of the tests indicate that the implementations are working correctly to the AES specification and therefore carrying out the encryption correctly.

10.2 Program Functionality

Since the application has an extremely simple user interface the number of different elements that can be tested is very limited. As the interface amounts to a single command this section of testing is only concerned with testing the invalid syntax handling of the program and mistakes made by the user when entering data. To achieve this, a number of different circumstances are simulated with the user input. The results to these simulations are shown in the following table.

Action	**Resulting Output**	**Comments**
Running the program with a valid set of arguments	Program returns the correct ciphertext.	This works correctly with all variations of plaintext and keys as shown in (10.1)
Just entering the command **AES** or **QAES** to the command prompt	`syntax: AES [key size in bits] [plaintext (32 Hex Chars)] [key]`	The application correctly stops the process and displays syntax help
Enter command missing the argument which specifies the key size	`syntax: AES [key size in bits] [plaintext (32 Hex Chars)] [key]`	Program gives guidance of how it should be used.
Enter command missing the argument which specifies the key size	`syntax: AES [key size in bits] [plaintext (32 Hex Chars)] [key]`	Program once again stops the operation and provides help
Enter command missing the argument which specifies the key size	`syntax: AES [key size in bits] [plaintext (32 Hex Chars)] [key]`	As above
Entering a key size different from 128, 192 or 256	`Invalid Key Size.  Must be 128, 192 or 256. Type AES for syntax help`	Application notices fault and stops processing. Also shows user how to get syntax help
Entering a plaintext which is too short or too long.	`Invalid length of plaintext or key, Type AES for syntax help`	Mistake is recognised and program stopped. Once again explains how to get syntax help.
Entering a key which does not match the given value in the key length argument.	`Invalid length of plaintext or key, Type AES for syntax help`	As Above
Having the key size argument in the wrong place	`Invalid Key Size.  Must be 128, 192 or 256. Type AES for syntax help`	As Above
Plaintext and Key orders being swapped round when key length is not equal to 128 bit.	`Invalid length of plaintext or key, Type AES for syntax help`	Program thinks that the plaintext has been entered incorrectly and so stops. **However the output does not help the user identify the real problem**

Action	**Resulting Output**	**Comments**
Plaintext and Key orders being swapped round when key length is equal to 128 bit.	Program returns the result believing that no mistake has been made.	**Since both the key and the plaintext are of the same format it is impossible for the program to recognise the incorrect input**.
Entering 3 characters instead of numbers for the key size	`Invalid Key Size.  Must be 128, 192 or 256. Type AES for syntax help`	The program recognises that the characters are not any of the allowed choices.
Entering non hexadecimal characters into key or plaintext.	A cipher text is returned	**The program has not picked up the problem. However it manages to continue without crashing**
Entering spurious characters instead of hex digits in key or plaintext but keeping the correct length	As Above	**As Above**

Table 5 : Test results of the simulated user inputs

As visible from the above table the majority of incorrect inputs are dealt with correctly by the program and in these cases the program offers help with how to enter the values in correctly. However a number of situations have arisen where the input checks have not recognised wrongly formatted inputs and have continued with the encryption process regardless.

Chapter 11

Speed Analysis

One of the main objectives for this dissertation is to conduct analysis of the speed at which the encryption can take place using the different implementations. This analysis can be used to help build an understanding of which hardware conditions suit different implementations.

As mentioned earlier we generally expect to achieve faster results from the high speed implementation due to the use of a number of lookup tables. However, the initialisation of these tables may produce the reverse effects when encrypting small amounts of data.

11.1 Method of Testing

The execution time of the programs will be conducted using the bash shell's `time` command on the following machine specification.

- Sun Ultra E480
- 4x900 Mhz UltraSparc III CPUs
- 16Gbytes RAM
- SunOS Solaris v9

The `time` command is only accurate to the nearest 10 milliseconds and so to create a more accurate solution a loop is placed in the source codes around the whole encryption process to force the program to run 100 times. Once the results are collected the resulting times are divided by 100 to achieve a real-time reading.

Each implementation (QAES and AES) is split three groups to accommodate the three different possible key sizes. Different data input sizes are then simulated on each these groups and timed using the above method. This is achieved using another loop placed in the source code around the parts of code which carry out the actual encryption algorithm only and not the 'initialisation' parts. This 'initialisation' part of code includes the key expansion algorithm since the same key would be used on the whole input. In addition to this, making the lookup tables in the high speed implementation would also fall into this category.

For example if we are trying to simulate a data input size of 64 bytes then the encryption process must be run a total of 4 times since 4 multiplied by 128 bits (the input plaintext size) is equal to 64 bytes

Each of these different input sizes is then tested a total of 5 times using different plaintext and key inputs taken from the first 5 values of the precompiled test data used earlier. The mean average of these times is then taken and used for comparison with other input sizes.

11.2 Results

The complete set of test data can be found in the appendix.

A summary of this data is in the following table. The values shown are the average execution times in seconds.

	AES 128	AES 192	AES 256	QAES 128	QAES 192	QAES 256
16 bytes	0.00858	0.00948	0.01170	0.01084	0.01084	0.01094
32 bytes	0.01518	0.01752	0.02100	0.01100	0.01090	0.01100
64 bytes	0.02866	0.03360	0.03966	0.01112	0.01098	0.01112
128 bytes	0.05560	0.06594	0.07774	0.01138	0.01136	0.01156
256 bytes	0.10994	0.12984	0.15218	0.01188	0.01204	0.01218
512 bytes	0.21810	0.25866	0.30184	0.01290	0.01322	0.01368
1 kilobyte	0.43370	0.51438	0.60180	0.01488	0.01566	0.01644

Table 6 : The execution times for the different implementations

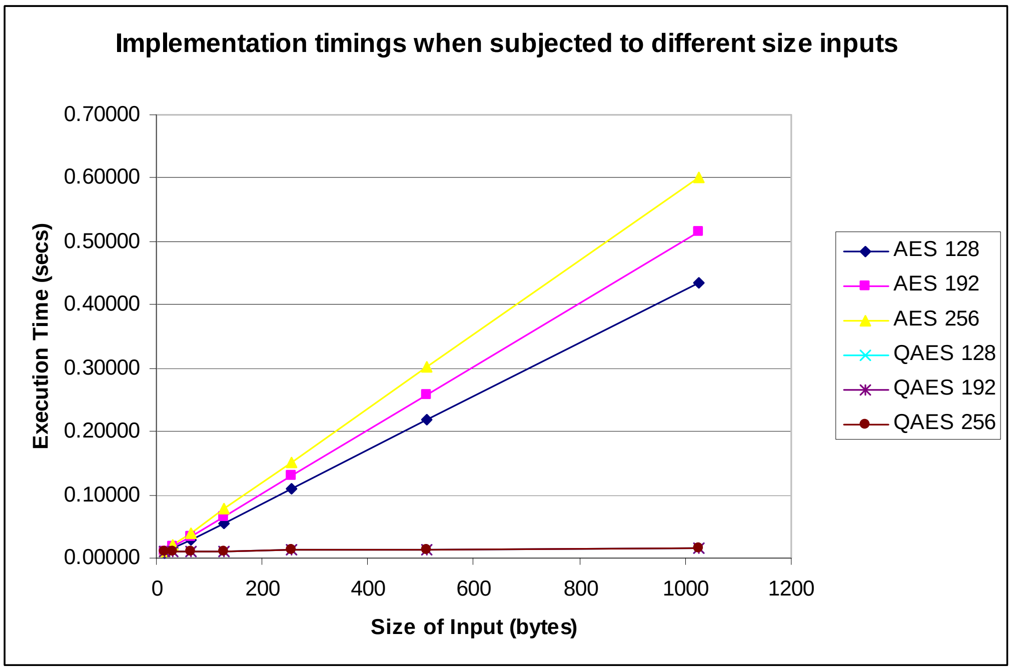


Figure 11: Chart showing the results of the above table

11.3 Analysis

As is to be expected the growth of both implementations is linear with respect to the input size. This makes sense since every time an extra 16 bytes is added to the input the algorithm has to encrypt another 128 bits of plaintext.

What is also clear from the graph is the effect of the key size on the speed of the algorithm using the low memory solution. What is not at first obvious is the fact that this same pattern is also evident in the high speed implementation values but the rate of growth for all three key lengths is so slow that it is not clearly visible from the graph.

What is interesting from the table is that the low memory implementation is actually slightly faster at encrypting a single 128 bit plaintext when the key size is equal to 128 and 192 bit. This means that the high speed implementation is not getting any gain from using the lookup tables. From the values it is possible to estimate the time it took to initialise the lookup tables as being approximately 0.1 seconds. Unlike with the low memory version where doubling the input size roughly doubles the time, the high speed version appears to increase in time very slowly after this initial 0.1 seconds.

Once the input size increases past the minimum 128 bits a clear advantage can be seen from using the lookup tables. Even up to the maximum 1 kilobyte tested the execution time had not doubled the initial 0.1 seconds taken by the initialisation process. This makes the high speed implementation very suitable for use in high volume encryptions. This is in contrast to the low memory solution where the time is increasing at a much faster rate. In fact when testing with 1 kilobyte of data the resulting execution time was about 50 times the original 128 bit speed.

From these figures it is clear to see the advantage of using the high speed version where the amount of data to is likely to exceed 128 bit. In fact, since the times are so close for even the initial 128 bit input test, it is fair to say that if the extra 1 kilobyte of memory space is available then the is no justification for using the low memory solution over the lookup tables for an AES encryption algorithm.

However, as unlikely as it is, if the circumstances are such that the algorithm will only ever need to encrypt 128 bits of plaintext using the same key then there is no reason why the low memory solution should not be implemented even if there is free memory space available. It is possible that his situation could arise in smart cards, where a just a personal identification number or a password may be encrypted.

Obviously if there is not enough memory space to be able to use lookup tables then there is only one choice of implementation that can be used. However, caution must be taken with the volume of data that needs to be encrypted with the same key since the computation time will continue to grow at the same rapid pace as is displayed above.

Chapter 12

Conclusion

12.1 Achievements

From the very start of this project it was expected that the task of implementing an encryption algorithm would be complex and this assumption was proved correct. The levels of mathematical complexity in the algorithm meant that a large amount of time was required to fully understand the structure of the cipher and the individual transformations involved in it before any form of implementation could be considered.

Fortunately there has been a reasonable amount of material available, each with a different person’s perspective, concerned with the AES algorithm. These different points of view help to build a good overall understanding and have been essential in achieving the desired goals.

The original objectives for this dissertation were to produce two fully working implementations of the AES encryption both with their own specific optimisations. It is evident from the test data that both implementations are encrypting correctly with respect to the official specification and therefore the first main objective has been met. However during the software testing a small number of problems arose where the program accepted non-hexadecimal characters as inputs when it should not. If the implementations are to be used in the future then a safety check must be placed at the start of the program to fix this problem.

In addition to this primary goal, there has been an extremely effective use of previous work to produce the optimisations that were required for each implementation to achieve its own individual objectives. The use of lookup tables in the high speed implementation has dramatically affected the time it takes to encrypt large amounts of data whilst still keeping the overheads of the initialisation process to a reasonable level. Furthermore, the on-the-fly techniques incorporated into the low memory solution has resulted in an implementation that can encrypt small amounts of data at competitive speeds, without the need for the additional memory space.

Hopefully the analysis of the timing results between the contrasting implementations will provide the AES community with a useful source of information which can be used to help judge which techniques are most suitable for their own personal specifications.

Possibly the most interesting results to have come from the analysis is the fact that on-the-fly solutions are actually capable of producing quicker results for encrypting small amounts of data. During the initial research conducted at the start of this dissertation there was little or no mention of this fact, presumably because most previous implementations assumed that the algorithm would never be used for such small inputs. However, I feel this is a valid result to state since there does exists situations where the minimum 16 bytes is all that needs encrypting using the same key.

An area of the project which I feel requires further work is the way the expanded key is calculated. Both implementations use the same function to achieve this expansion which requires a [4 x 60] byte array to hold the calculated values. During the design of the algorithm I was unable to find or invent a method for calculating the round keys on-the-fly which would have released the 240 bytes used for holding the entire expanded key. This would have produced a superior low memory implementation than the one created.

A discussion of possible extensions to the achieved work was discussed in chapter 4. Unfortunately, due to the time taken to achieve the primary goals and making sure there was enough time to write up this dissertation, these extra goals were left unattempted. One of these extensions included implementing the equivalent decryption algorithms to pair up with the two encryption algorithms. This would require using the same techniques adopted in the encryption implementations with the corresponding inverse transformations needed in order to decrypt a ciphertext.

12.2 Acquired Knowledge

Undertaking this project has opened up a whole new area of computing and mathematics to me. The majority of the fundamental mathematics that I have learnt while doing this project is directly applicable to other methods of cryptography. Also, the use of finite fields in computing is always going to be a necessity since there will only ever be a finite number of elements that can be represented on a computer system.

In addition to this fundamental mathematics I have also learned much about the ways in which ciphers are constructed, the need for cipher attributes such as diffusion and also an understanding of cryptanalysis techniques used to test the strengths and weaknesses in encryption algorithms.

12.3 Further Work

The beauty of implementing two very different algorithms is the ability to produce many more versions using a combination of techniques taken from the original two.

For instance, if someone wishing to create an AES implementation had certain memory constraints which would not allow the full use of lookup tables then a simple mixture of the two versions could prove to be a good compromise.

However, to produce an efficient algorithm using this technique, it would be wise to take the time to analyse which lookup tables should be kept and which should be replaced by on-the-fly operations to result in the best possible speed increases. Basically this requires calculating which lookup table produces the largest reduction in computational time per unit increase of memory space.

In order to carry out this analysis there would be a number of different variables that would need to be taken into consideration. The first variable would be the number of times each operation is actually required in one full run of the encryption process. This could be calculated using a number of counters placed inside the specific operations to calculate the total number of calls made to each one during the entire process.

Another variable to consider would be the number and size of tables required to replace the operation using lookup tables. Finally the complexity of the on-the-fly operation would also have to be calculated as this will directly effect the time it takes to complete the encryption

The analysis of these variables directly follows on from the work produced in this dissertation and is just one example of how this work can be expanded in future articles.

12.4 Final Comments

This dissertation should be perceived as an overall success not only due to the successful implementation of two different optimised versions of the AES, algorithm but also for the useful insight that the speed analysis has produced. This was only possible through the availability of high quality previous work on AES topics.

I now plan to use the implementations created here within a larger program that will have the ability to encode whole data files so they can be safely transported across the internet. I have full confidence that the encryption algorithms produced for this dissertation will be up to the job.

Chapter 13

Bibliography

Anon (2001), "Announcing the Advanced Encryption Standard (AES)" *Federal Information Standards Publication 197* – November 2001

Becket, B. (1988), "Introduction to Cryptology", *Blackwell Scientific Publications*

Beker, H., Piper, F. (1982), "Cipher Systems – The protection of communications", *Northwood Publications*

Biham, E. (1998), "DES, Triple-DES and AES", *CryptoBytes Technical Newsletter – RSA Laboratories,* Volume 4 – Summer 1998

Daemen, J., Rijmen, V. (1999), "AES Proposal: Rijndael", available from http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

Daemen, J., Rijmen, V. (2000), "Rijndael for AES", *The Third Advanced Encryption Standard Candidate Conference,* April 13-14, 2000, New York

Daemen, J., Rijmen, V. (2002), "The Design of Rijndael, AES – The advanced Encryption Standard", *Springer-Verlag Publications*

Ferguson, N., Kelsey, J., Lucks, S., Schneier, B., Stay, M., Wagner, D., Whiting, D. (2002), "Improved Cryptanalysis of Rijndael", *Seventh Fast Software Encryption Workshop*, Springer-Verlag, 2000

Gladman, B. (2003), "A Specification for The AES Algorithm", available from http://fp.gladman.plus.com/cryptography_technology/rijndael/aes.spec.311.pdf

Hankerson, D., Menezes, A., Vanstone, S. (2003), "Guide to Elliptic Curve Cryptography", *Springer Publications*, December 2003, ISBN: 0-387-95273-X

Lane, V.P. (1985), "Security of Computer Based Information Systems", *Macmillan Education LTD*

Koblitz, N.(1998), “Algebraic Aspects of Cryptography” *Springer Publications*

Rijmen, V. (2000) , “Efficient Implementation of the Rijndael S-box”, *NIST's AES home page*, http://www.nist.gov/aes

Seberry, J., Pierprzyk, J. (1989), “An Introduction to Computer Security” *Prentice Hall – Advances in Computer Science Series*

Standaert, F., Rouvroy, G., Quisquater, J., Legat, J. (2003), “Applications: A methodology to implement block ciphers in reconfigurable hardware and its application to fast and compact AES RIJNDAEL”
Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays.

Ward, R., Molteno, T. (2003), “Efficient Hardware Calculation of Inverses in GF(2^8)”, *Proceedings of ENZCon'03, University of Waikato, NZ*, September 2003.

Welsh, D. (1988), “Codes and Cryptography”, *Oxford Science Publications*

Wiener, M.J. (1996), “Efficient DES Key Search”, Presented at the Rump session of CRYPTO’93. Reprinted in *Practical Cryptography for Data Internetworks*, IEEE Computer Society Press

(url: crypto)
Cryptomathic - DO WE NEED AES? [online] – Available from: http://www.cryptomathic.com/company/aes.html
[accessed: November 2004]

(url: TheAES)
The Advanced Encryption Standard (Rijndael) [online] – Available from: http://csrc.nist.gov/CryptoToolkit/aes/
[accessed: November 2004]

(url: NIST)
National Institute for Standards and Technology? [online] – Available from: http://www.cryptomathic.com/company/aes.html
[accessed: November 2004]

Chapter 14

Appendices

A Installation Guide

The following instructions explain how to use the source files (AES.c and QAES.c) to create a working application and how to use this application to achieve a desired encryption.

These instructions are for use on a UNIX terminal and use the C compiler `gcc`. In order to use a different compiler please refer to the relevant UNIX manual page for guidance.

1. Create a new directory called AES

2. Copy the files ‘QAES.c’ and ‘AES.c’ into the new directory

3. Compile the two programs using the following commands:

```
gcc AES.c -o AES -lm
gcc QAES.c -o QAES -lm
```

4. To run the programs use one of the following commands depending on the required version:

```
AES [key size in bits] [plaintext (32 chars)] [key]
QAES [key size in bits] [plaintext (32 chars)] [key]
```

For example, using the high speed implementation, and with

Key = “000102030405060708090A0B0C0D0E0F” (128 bits)
Plaintext = “FEDCBA98765432100123456789ABCDEF”

Then the command would be:

```
QAES 128 FEDCBA98765432100123456789ABCDEF 000102030405060708090A0B0C0D0E0F
```

B Speed Analysis Test Results

The following tables hold the complete results from the application speed tests used in chapter 11.

Note that the ‘actual time’ refers to the average time divided by 100 since during the testing the program is looped 100 times to produce more accurate results.

All time values are in seconds.

Low Memory Implementation (128 bit key length)

Data Input Size (bits)	Time 1	Time 2	Time 3	Time 4	Time 5	Average Time	Actual Time
128	0.86	0.85	0.86	0.86	0.86	0.86	**0.00858**
256	1.52	1.53	1.52	1.50	1.52	1.52	**0.01518**
512	2.87	2.86	2.86	2.87	2.87	2.87	**0.02866**
1024	5.53	5.57	5.56	5.57	5.57	5.56	**0.05560**
2048	10.93	10.96	11.14	10.96	10.98	10.99	**0.10994**
4096	21.80	21.66	21.75	22.00	21.84	21.81	**0.21810**
8192	43.46	43.31	43.27	43.23	43.58	43.37	**0.43370**

Low Memory Implementation (192 bit key length)

Data Input Size (bits)	Time 1	Time 2	Time 3	Time 4	Time 5	Average Time	Actual Time
128	0.95	0.95	0.94	0.95	0.95	0.95	**0.00948**
256	1.75	1.76	1.75	1.75	1.75	1.75	**0.01752**
512	3.35	3.38	3.36	3.36	3.35	3.36	**0.03360**
1024	6.54	6.56	6.56	6.70	6.61	6.59	**0.06594**
2048	13.00	12.96	12.99	13.00	12.97	12.98	**0.12984**
4096	25.77	25.98	25.85	25.87	25.86	25.87	**0.25866**
8192	51.49	51.46	51.38	51.46	51.40	51.44	**0.51438**

Low Memory Implementation (256 bit key length)

Data Input Size (bits)	Time 1	Time 2	Time 3	Time 4	Time 5	Average Time	Actual Time
128	1.17	1.18	1.18	1.15	1.17	1.17	**0.01170**
256	2.10	2.09	2.12	2.10	2.09	2.10	**0.02100**
512	3.97	3.97	3.96	3.97	3.96	3.97	**0.03966**
1024	7.99	7.72	7.69	7.73	7.74	7.77	**0.07774**
2048	15.19	15.21	15.22	15.26	15.21	15.22	**0.15218**
4096	30.10	30.14	30.10	30.17	30.41	30.18	**0.30184**
8192	60.13	60.08	60.07	60.47	60.15	60.18	**0.60180**

High Speed Implementation (128 bit key length)

Data Input Size (bits)	Time 1	Time 2	Time 3	Time 4	Time 5	Average Time	Actual Time
128	1.09	1.09	1.08	1.08	1.08	1.08	**0.01084**
256	1.10	1.11	1.11	1.09	1.09	1.10	**0.01100**
512	1.11	1.11	1.11	1.12	1.11	1.11	**0.01112**
1024	1.14	1.13	1.14	1.14	1.14	1.14	**0.01138**
2048	1.20	1.19	1.19	1.18	1.18	1.19	**0.01188**
4096	1.29	1.29	1.28	1.30	1.29	1.29	**0.01290**
8192	1.48	1.49	1.49	1.48	1.50	1.49	**0.01488**

High Speed Implementation (192 bit key length)

Data Input Size (bits)	Time 1	Time 2	Time 3	Time 4	Time 5	Average Time	Actual Time
128	1.08	1.09	1.10	1.08	1.07	1.08	**0.01084**
256	1.09	1.09	1.09	1.09	1.09	1.09	**0.01090**
512	1.10	1.10	1.08	1.10	1.11	1.10	**0.01098**
1024	1.14	1.14	1.14	1.13	1.13	1.14	**0.01136**
2048	1.21	1.22	1.21	1.18	1.20	1.20	**0.01204**
4096	1.32	1.32	1.32	1.32	1.33	1.32	**0.01322**
8192	1.56	1.57	1.56	1.56	1.58	1.57	**0.01566**

High Speed Implementation (256 bit key length)

Data Input Size (bits)	Time 1	Time 2	Time 3	Time 4	Time 5	Average Time	Actual Time
128	1.11	1.08	1.09	1.09	1.10	1.09	**0.01094**
256	1.12	1.10	1.09	1.09	1.10	1.10	**0.01100**
512	1.11	1.10	1.11	1.12	1.12	1.11	**0.01112**
1024	1.15	1.15	1.15	1.17	1.16	1.16	**0.01156**
2048	1.23	1.21	1.21	1.22	1.22	1.22	**0.01218**
4096	1.38	1.36	1.37	1.36	1.37	1.37	**0.01368**
8192	1.65	1.65	1.66	1.63	1.63	1.64	**0.01644**

C Encryption Accuracy Test Results

In order to test a large volume of different plaintexts and keys a UNIX bash shell script was implemented. This script asks for the filename of the test values and the name of the implementation and carries out the bulk encryptions.

The code for this script is as follows:

```
#!/bin/bash

(rm testdata.out)

echo -n "File to read test values from? "
read file

echo -n "Which AES program? "
read command

while read line
do

      X="$command $line"
      echo $X
      ($X) >> testdata.out

done <$file
```

To run the script the following command must be entered into UNIX.

```
bash testscript
```

An example of how the script is run on the machine ‘amos’ is shown below.

```
      amos $ bash testscript
      File to read test values from? values
      Which AES program? QAES
```

The full set of test results referred to in chapter 10 is available on the following pages.

Test results for Fast Implementation (AES) - 128bit Key

Plaintext	Key	Expected Output	Actual Output	Pass?
506812A45F08C889B97F5980038B8359	00010203050607080A0B0C0D0F101112	D8F532538289EF7D06B506A4FD5BE9C9	D8F532538289EF7D06B506A4FD5BE9C9	Pass
5C6D71CA30DE8B8B00549984D2EC7D4B	14151617191A1B1C1E1F202123242526	59AB30F4D4EE6E4FF9907EF65B1FB68C	59AB30F4D4EE6E4FF9907EF65B1FB68C	Pass
53F3F4C64F8616E4E7C56199F48F21F6	28292A2B2D2E2F30323334353738393A	BF1ED2FCB2AF3FD41443B56D85025CB1	BF1ED2FCB2AF3FD41443B56D85025CB1	Pass
A1EB65A3487165FB0F1C27FF9959F703	3C3D3E3F41424344464748494B4C4D4E	7316632D5C32233EDCB0780560EAE8B2	7316632D5C32233EDCB0780560EAE8B2	Pass
3553ECF0B1739558B08E350A98A39BFA	50515253555657585A5B5C5D5F606162	408C073E3E2538072B72625E68B8364B	408C073E3E2538072B72625E68B8364B	Pass
67429969490B9711AE2B01DC497AFDE8	64656667696A6B6C6E6F707173747576	E1F94DFA776597BEACA262F2F6366FEA	E1F94DFA776597BEACA262F2F6366FEA	Pass
93385C1F2AEC8BED192F5A8E161DD508	78797A7B7D7E7F80828384858788898A	F29E986C6A1C27D7B29FFD7EE92B75F1	F29E986C6A1C27D7B29FFD7EE92B75F1	Pass
B5BF946BE19BEB8DB3983B5F4C6E8DDB	8C8D8E8F91929394969798999B9C9D9E	131C886A57F8C2E713ABA6955E2B55B5	131C886A57F8C2E713ABA6955E2B55B5	Pass
41321EE10E21BD907227C4450FF42324	A0A1A2A3A5A6A7A8AAABACADAFB0B1B2	D2AB7662DF9B8C740210E5EEB61C199D	D2AB7662DF9B8C740210E5EEB61C199D	Pass
00A82F59C91C8486D12C0A80124F6089	B4B5B6B7B9BABBBCBEBFC0C1C3C4C5C6	14C10554B2859C484CAB5869BBE7C470	14C10554B2859C484CAB5869BBE7C470	Pass
7CE0FD076754691B4BBD9FAF8A1372FE	C8C9CACBCDCECFD0D2D3D4D5D7D8D9DA	DB4D498F0A49CF55445D502C1F9AB3B5	DB4D498F0A49CF55445D502C1F9AB3B5	Pass
23605A8243D07764541BC5AD355B3129	DCDDDEDFE1E2E3E4E6E7E8E9EBECEDEE	6D96FEF7D66590A77A77BB2056667F7F	6D96FEF7D66590A77A77BB2056667F7F	Pass
12A8CFA23EA764FD876232B4E842BC44	F0F1F2F3F5F6F7F8FAFBFCFDFE010002	316FB68EDBA736C53E78477BF913725C	316FB68EDBA736C53E78477BF913725C	Pass
BCAF32415E8308B3723E5FDD853CCC80	04050607090A0B0C0E0F101113141516	6936F2B93AF8397FD3A771FC011C8C37	6936F2B93AF8397FD3A771FC011C8C37	Pass
89AFAE685D801AD747ACE91FC49ADDE0	2C2D2E2F31323334363738393B3C3D3E	F3F92F7A9C59179C1FCC2C2BA0B082CD	F3F92F7A9C59179C1FCC2C2BA0B082CD	Pass
F521D07B484357C4A69E76124A634216	40414243454647484A4B4C4D4F505152	6A95EA659EE3889158E7A9152FF04EBC	6A95EA659EE3889158E7A9152FF04EBC	Pass
3E23B3BC065BCC152407E23896D77783	54555657595A5B5C5E5F606163646566	1959338344E945670678A5D432C90B93	1959338344E945670678A5D432C90B93	Pass
79F0FBA002BE1744670E7E99290D8F52	68696A6B6D6E6F70727374757778797A	E49BDDD2369B83EE66E6C75A1161B394	E49BDDD2369B83EE66E6C75A1161B394	Pass
DA23FE9D5BD63E1D72E3DAFBE21A6C2A	7C7D7E7F81828384868788898B8C8D8E	D3388F19057FF704B70784164A74867D	D3388F19057FF704B70784164A74867D	Pass
E3F5698BA90B6A022EFD7DB2C7E6C823	A4A5A6A7A9AAABACAEAFB0B1B3B4B5B6	23AA03E2D5E4CD24F3217E596480D1E1	23AA03E2D5E4CD24F3217E596480D1E1	Pass
BDC2691D4F1B73D2700679C3BCBF9C6E	E0E1E2E3E5E6E7E8EAEBECEDEFF0F1F2	C84113D68B666AB2A50A8BDB222E91B9	C84113D68B666AB2A50A8BDB222E91B9	Pass
BA74E02093217EE1BA1B42BD5624349A	08090A0B0D0E0F10121314151718191A	AC02403981CD4340B507963DB65CB7B6	AC02403981CD4340B507963DB65CB7B6	Pass

B5C593B5851C57FBF8B3F57715E8F680	6C6D6E6F71727374767778797B7C7D7E	8D1299236223359474011F6BF5088414	8D1299236223359474011F6BF5088414	Pass
3DA9BD9CEC072381788F9387C3BBF4EE	80818283858687888A8B8C8D8F909192	5A1D6AB8605505F7977E55B9A54D9B90	5A1D6AB8605505F7977E55B9A54D9B90	Pass
4197F3051121702AB65D316B3C637374	94959697999A9B9C9E9FA0A1A3A4A5A6	72E9C2D519CF555E4208805AABE3B258	72E9C2D519CF555E4208805AABE3B258	Pass
9F46C62EC4F6EE3F6E8C62554BC48AB7	A8A9AAABADAEAFB0B2B3B4B5B7B8B9BA	A8F3E81C4A23A39EF4D745DFFE026E80	A8F3E81C4A23A39EF4D745DFFE026E80	Pass
0220673FE9E699A4EBC8E0DBEB6979C8	BCBDBEBFC1C2C3C4C6C7C8C9CBCCCDCE	546F646449D31458F9EB4EF5483AEE6C	546F646449D31458F9EB4EF5483AEE6C	Pass
B2B99171337DED9BC8C2C23FF6F18867	D0D1D2D3D5D6D7D8DADBDCDDDFE0E1E2	4DBE4BC84AC797C0EE4EFB7F1A07401C	4DBE4BC84AC797C0EE4EFB7F1A07401C	Pass
A7FACF4E301E984E5EFEEFD645B23505	E4E5E6E7E9EAEBECEEEFF0F1F3F4F5F6	25E10BFB411BBD4D625AC8795C8CA3B3	25E10BFB411BBD4D625AC8795C8CA3B3	Pass
F7C762E4A9819160FD7ACFB6C4EEDCDD	F8F9FAFBFDFEFE00020304050708090A	315637405054EC803614E43DEF177579	315637405054EC803614E43DEF177579	Pass
9B64FC21EA08709F4915436FAA70F1BE	0C0D0E0F11121314161718191B1C1D1E	60C5BC8A1410247295C6386C59E572A8	60C5BC8A1410247295C6386C59E572A8	Pass
52AF2C3DE07EE6777F55A4ABFC100B3F	2021222325262728 2A2B2C2D2F303132	01366FC8CA52DFE055D6A00A76471BA6	01366FC8CA52DFE055D6A00A76471BA6	Pass
2FCA001224386C57AA3F968CBE2C816F	34353637393A3B3C3E3F404143444546	ECC46595516EC612449C3F581E7D42FF	ECC46595516EC612449C3F581E7D42FF	Pass
4149C73658A4A9C564342755EE2C132F	48494A4B4D4E4F50525354555758595A	6B7FFE4C602A154B06EE9C7DAB5331C9	6B7FFE4C602A154B06EE9C7DAB5331C9	Pass
AF60005A00A1772F7C07A48A923C23D2	5C5D5E5F61626364666768696B6C6D6E	7DA234C14039A240DD02DD0FBF84EB67	7DA234C14039A240DD02DD0FBF84EB67	Pass
6FCCBC28363759914B6F0280AFAF20C6	70717273757677787A7B7C7D7F808182	C7DC217D9E3604FFE7E91F080ECD5A3A	C7DC217D9E3604FFE7E91F080ECD5A3A	Pass
7D82A43DDF4FEFA2FC5947499884D386	84858687898A8B8C8E8F909193949596	37785901863F5C81260EA41E7580CDA5	37785901863F5C81260EA41E7580CDA5	Pass
5D5A990EAAB9093AFE4CE254DFA49EF9	98999A9B9D9E9FA0A2A3A4A5A7A8A9AA	A07B9338E92ED105E6AD720FCCCE9FE4	A07B9338E92ED105E6AD720FCCCE9FE4	Pass
4CD1E2FD3F4434B553AAE453F0ED1A02	ACADAEAFB1B2B3B4B6B7B8B9BBBCBDBE	AE0FB9722418CC21A7DA816BBC61322C	AE0FB9722418CC21A7DA816BBC61322C	Pass
5A2C9A9641D4299125FA1B9363104B5E	C0C1C2C3C5C6C7C8CACBCCCDCFD0D1D2	C826A193080FF91FFB21F71D3373C877	C826A193080FF91FFB21F71D3373C877	Pass
B517FE34C0FA217D341740BFD4FE8DD4	D4D5D6D7D9DADBDCDEDFE0E1E3E4E5E6	1181B11B0E494E8D8B0AA6B1D5AC2C48	1181B11B0E494E8D8B0AA6B1D5AC2C48	Pass
014BAF2278A69D331D5180103643E99A	E8E9EAEBEDEEEFF0F2F3F4F5F7F8F9FA	6743C3D1519AB4F2CD9A78AB09A511BD	6743C3D1519AB4F2CD9A78AB09A511BD	Pass
B529BD8164F20D0AA443D4932116841C	FCFDFEFF01020304060708090B0C0D0E	DC55C076D52BACDF2EEFD952946A439D	DC55C076D52BACDF2EEFD952946A439D	Pass
2E596DCBB2F33D4216A1176D5BD1E456	10111213151617181A1B1C1D1F202122	711B17B590FFC72B5C8E342B601E8003	711B17B590FFC72B5C8E342B601E8003	Pass
7274A1EA2B7EE2424E9A0E4673689143	24252627292A2B2C2E2F303133343536	19983BB0950783A537E1339F4AA21C75	19983BB0950783A537E1339F4AA21C75	Pass
AE20020BD4F13E9D90140BEE3B5D26AF	38393A3B3D3E3F40424344454748494A	3BA7762E15554169C0F4FA39164C410C	3BA7762E15554169C0F4FA39164C410C	Pass
BAAC065DA7AC26E855E79C8849D75A02	4C4D4E4F51525354565758595B5C5D5E	A0564C41245AFCA7AF8AA2E0E588EA89	A0564C41245AFCA7AF8AA2E0E588EA89	Pass
7C917D8D1D45FAB9E2540E28832540CC	60616263656667686A6B6C6D6F707172	5E36A42A2E099F54AE85ECD92E2381ED	5E36A42A2E099F54AE85ECD92E2381ED	Pass
BDE6F89E16DAADB0E847A2A614566A91	74757677797A7B7C7E7F808183848586	770036F878CD0F6CA2268172F106F2FE	770036F878CD0F6CA2268172F106F2FE	Pass
C9DE163725F1F5BE44EBB1DB51D07FBC	88898A8B8D8E8F90929394959798999A	7E4E03908B716116443CCF7C94E7C259	7E4E03908B716116443CCF7C94E7C259	Pass

Test results for Fast Implementation (AES) - 192bit Key

Plaintext	Key	Expected Result	Actual Result	Pass?
2D33EEF2C0430A8A9EBF45E809C40BB6	00010203050607080A0B0C0D0F10111214151617191A1B1C	DFF4945E0336DF4C1C56BC700EFF837F	DFF4945E0336DF4C1C56BC700EFF837F	Pass
6AA375D1FA155A61FB72353E0A5A8756	1E1F20212324252628292A2B2D2E2F30323334353738393A	B6FDDEF4752765E347D5D2DC196D1252	B6FDDEF4752765E347D5D2DC196D1252	Pass
BC3736518B9490DCB8ED60EB26758ED4	3C3D3E3F41424344464748494B4C4D4E5051525355565758	D23684E3D963B3AFCF1A114ACA90CBD6	D23684E3D963B3AFCF1A114ACA90CBD6	Pass
AA214402B46CFFB9F761EC11263A311E	5A5B5C5D5F60616264656667696A6B6C6E6F707173747576	3A7AC027753E2A18C2CEAB9E17C11FD0	3A7AC027753E2A18C2CEAB9E17C11FD0	Pass
02AEA86E572EEAB66B2C3AF5E9A46FD6	78797A7B7D7E7F80828384858788898A8C8D8E8F91929394	8F6786BD007528BA26603C1601CDD0D8	8F6786BD007528BA26603C1601CDD0D8	Pass
E2AEF6ACC33B965C4FA1F91C75FF6F36	969798999B9C9D9EA0A1A2A3A5A6A7A8AAABACADAFB0B1B2	D17D073B01E71502E28B47AB551168B3	D17D073B01E71502E28B47AB551168B3	Pass
0659DF46427162B9434865DD9499F91D	B4B5B6B7B9BABBBCBEBFC0C1C3C4C5C6C8C9CACBCDCECFD0	A469DA517119FAB95876F41D06D40FFA	A469DA517119FAB95876F41D06D40FFA	Pass
49A44239C748FEB456F59C276A5658DF	D2D3D4D5D7D8D9DADCDDDEDFE1E2E3E4E6E7E8E9EBECEDEE	6091AA3B695C11F5C0B6AD26D3D862FF	6091AA3B695C11F5C0B6AD26D3D862FF	Pass
66208F6E9D04525BDEDB2733B6A6BE37	F0F1F2F3F5F6F7F8FAFBFCFDFE01000204050607090A0B0C	70F9E67F9F8DF1294131662DC6E69364	70F9E67F9F8DF1294131662DC6E69364	Pass
3393F8DFC729C97F5480B950BC9666B0	0E0F10111314151618191A1B1D1E1F20222324252728292A	D154DCAFAD8B207FA5CBC95E9996B559	D154DCAFAD8B207FA5CBC95E9996B559	Pass
606834C8CE063F3234CF1145325DBD71	2C2D2E2F31323334363738393B3C3D3E4041424345464748	4934D541E8B46FA339C805A7AEB9E5DA	4934D541E8B46FA339C805A7AEB9E5DA	Pass
FEC1C04F529BBD17D8CECFCC4718B17F	4A4B4C4D4F50515254555657595A5B5C5E5F606163646566	62564C738F3EFE186E1A127A0C4D3C61	62564C738F3EFE186E1A127A0C4D3C61	Pass
32DF99B431ED5DC5ACF8CAF6DC6CE475	68696A6B6D6E6F70727374757778797A7C7D7E7F81828384	07805AA043986EB23693E23BEF8F3438	07805AA043986EB23693E23BEF8F3438	Pass
7FDC2B746F3F665296943B83710D1F82	868788898B8C8D8E90919293959697989A9B9C9D9FA0A1A2	DF0B4931038BADE848DEE3B4B85AA44B	DF0B4931038BADE848DEE3B4B85AA44B	Pass
8FBA1510A3C5B87E2EAA3F7A91455CA2	A4A5A6A7A9AAABACAEAFB0B1B3B4B5B6B8B9BABBBDBEBFC0	592D5FDED76582E4143C65099309477C	592D5FDED76582E4143C65099309477C	Pass
2C9B468B1C2EED92578D41B0716B223B	C2C3C4C5C7C8C9CACCCDCECFD1D2D3D4D6D7D8D9DBDCDDDE	C9B8D6545580D3DFBCDD09B954ED4E92	C9B8D6545580D3DFBCDD09B954ED4E92	Pass
0A2BBF0EFC6BC0034F8A03433FCA1B1A	E0E1E2E3E5E6E7E8EAEBECEDEFF0F1F2F4F5F6F7F9FAFBFC	5DCCD5D6EB7C1B42ACB008201DF707A0	5DCCD5D6EB7C1B42ACB008201DF707A0	Pass
25260E1F31F4104D387222E70632504B	FEFE01010304050608090A0B0D0E0F10121314151718191A	A2A91682FFEB6ED1D34340946829E6F9	A2A91682FFEB6ED1D34340946829E6F9	Pass
C527D25A49F08A5228D338642AE65137	1C1D1E1F21222324262728292B2C2D2E3031323335363738	E45D185B797000348D9267960A68435D	E45D185B797000348D9267960A68435D	Pass
3B49FC081432F5890D0E3D87E884A69E	3A3B3C3D3F40414244454647494A4B4C4E4F505153545556	45E060DAE5901CDA8089E10D4F4C246B	45E060DAE5901CDA8089E10D4F4C246B	Pass
D173F9ED1E57597E166931DF2754A083	58595A5B5D5E5F60626364656768696A6C6D6E6F71727374	F6951AFACC0079A369C71FDCFF45DF50	F6951AFACC0079A369C71FDCFF45DF50	Pass
8C2B7CAFA5AFE7F13562DAEAE1ADEDE0	767778797B7C7D7E80818283858687888A8B8C8D8F909192	9E95E00F351D5B3AC3D0E22E626DDAD6	9E95E00F351D5B3AC3D0E22E626DDAD6	Pass
AAF4EC8C1A815AEB826CAB741339532C	94959697999A9B9C9E9FA0A1A3A4A5A6A8A9AAABADAEAFB0	9CB566FF26D92DAD083B51FDC18C173C	9CB566FF26D92DAD083B51FDC18C173C	Pass

40BE8C5D9108E663F38F1A2395279ECF	D0D1D2D3D5D6D7D8DADBDCDDDFE0E1E2E4E5E6E7E9EAEBEC	C9C82766176A9B228EB9A974A010B4FB	C9C82766176A9B228EB9A974A010B4FB	Pass
0C8AD9BC32D43E04716753AA4CFBE351	2A2B2C2D2F30313234353637393A3B3C3E3F404143444546	D8E26AA02945881D5137F1C1E1386E88	D8E26AA02945881D5137F1C1E1386E88	Pass
1407B1D5F87D63357C8DC7EBBAEBBFEE	48494A4B4D4E4F50525354555758595A5C5D5E5F61626364	C0E024CCD68FF5FFA4D139C355A77C55	C0E024CCD68FF5FFA4D139C355A77C55	Pass
E62734D1AE3378C4549E939E6F123416	84858687898A8B8C8E8F90919394959698999A9B9D9E9FA0	0B18B3D16F491619DA338640DF391D43	0B18B3D16F491619DA338640DF391D43	Pass
5A752CFF2A176DB1A1DE77F2D2CDEE41	A2A3A4A5A7A8A9AAACADAEAFB1B2B3B4B6B7B8B9BBBCBDBE	DBE09AC8F66027BF20CB6E434F252EFC	DBE09AC8F66027BF20CB6E434F252EFC	Pass
A9C8C3A4EABEDC80C64730DDD018CD88	C0C1C2C3C5C6C7C8CACBCCCDCFD0D1D2D4D5D6D7D9DADBDC	6D04E5E43C5B9CBE05FEB9606B6480FE	6D04E5E43C5B9CBE05FEB9606B6480FE	Pass
EE9B3DBBDB86180072130834D305999A	1A1B1C1D1F20212224252627292A2B2C2E2F303133343536	DD1D6553B96BE526D9FEE0FBD7176866	DD1D6553B96BE526D9FEE0FBD7176866	Pass
A7FA8C3586B8EBDE7568EAD6F634A879	38393A3B3D3E3F40424344454748494A4C4D4E4F51525354	0260CA7E3F979FD015B0DD4690E16D2A	0260CA7E3F979FD015B0DD4690E16D2A	Pass
37E0F4A87F127D45AC936FE7AD88C10A	929394959798999A9C9D9E9FA1A2A3A4A6A7A8A9ABACADAE	9893734DE10EDCC8A67C3B110B8B8CC6	9893734DE10EDCC8A67C3B110B8B8CC6	Pass
3F77D8B5D92BAC148E4E46F697A535C5	464748494B4C4D4E50515253555657585A5B5C5D5F606162	93B30B750516B2D18808D710C2EE84EF	93B30B750516B2D18808D710C2EE84EF	Pass
D25EBB686C40F7E2C4DA1014936571CA	828384858788898A8C8D8E8F91929394969798999B9C9D9E	16F65FA47BE3CB5E6DFE7C6C37016C0E	16F65FA47BE3CB5E6DFE7C6C37016C0E	Pass
4F1C769D1E5B0552C7ECA84DEA26A549	A0A1A2A3A5A6A7A8AAABACADAFB0B1B2B4B5B6B7B9BABBBC	F3847210D5391E2360608E5ACB560581	F3847210D5391E2360608E5ACB560581	Pass
8548E2F882D7584D0FAFC54372B6633A	BEBFC0C1C3C4C5C6C8C9CACBCDCECFD0D2D3D4D5D7D8D9DA	8754462CD223366D0753913E6AF2643D	8754462CD223366D0753913E6AF2643D	Pass
87D7A336CB476F177CD2A51AF2A62CDF	DCDDDEDFE1E2E3E4E6E7E8E9EBECEDEEF0F1F2F3F5F6F7F8	1EA20617468D1B806A1FD58145462017	1EA20617468D1B806A1FD58145462017	Pass
03B1FEAC668C4E485C1065DFC22B44EE	FAFBFCFDFE01000204050607090A0B0C0E0F101113141516	3B155D927355D737C6BE9DDA60136E2E	3B155D927355D737C6BE9DDA60136E2E	Pass
BDA15E66819FA72D653A6866AA287962	18191A1B1D1E1F20222324252728292A2C2D2E2F31323334	26144F7B66DAA91B6333DBD3850502B3	26144F7B66DAA91B6333DBD3850502B3	Pass
4D0C7A0D2505B80BF8B62CEB12467F0A	363738393B3C3D3E40414243454647484A4B4C4D4F505152	E4F9A4AB52CED8134C649BF319EBCC90	E4F9A4AB52CED8134C649BF319EBCC90	Pass
626D34C9429B37211330986466B94E5F	54555657595A5B5C5E5F60616364656668696A6B6D6E6F70	B9DDD29AC6128A6CAB121E34A4C62B36	B9DDD29AC6128A6CAB121E34A4C62B36	Pass
333C3E6BF00656B088A17E5FF0E7F60A	727374757778797A7C7D7E7F81828384868788898B8C8D8E	6FCDDAD898F2CE4EFF51294F5EAAF5C9	6FCDDAD898F2CE4EFF51294F5EAAF5C9	Pass
687ED0CDC0D2A2BC8C466D05EF9D2891	90919293959697989A9B9C9D9FA0A1A2A4A5A6A7A9AAABAC	C9A6FE2BF4028080BEA6F7FC417BD7E3	C9A6FE2BF4028080BEA6F7FC417BD7E3	Pass
487830E78CC56C1693E64B2A6660C7B6	AEAFB0B1B3B4B5B6B8B9BABBBDBEBFC0C2C3C4C5C7C8C9CA	6A2026846D8609D60F298A9C0673127F	6A2026846D8609D60F298A9C0673127F	Pass
7A48D6B7B52B29392AA2072A32B66160	CCCDCECFD1D2D3D4D6D7D8D9DBDCDDDEE0E1E2E3E5E6E7E8	2CB25C005E26EFEA44336C4C97A4240B	2CB25C005E26EFEA44336C4C97A4240B	Pass
907320E64C8C5314D10F8D7A11C8618D	EAEBECEDEFF0F1F2F4F5F6F7F9FAFBFCFEFE010103040506	496967AB8680DDD73D09A0E4C7DCC8AA	496967AB8680DDD73D09A0E4C7DCC8AA	Pass
B561F2CA2D6E65A4A98341F3ED9FF533	08090A0B0D0E0F10121314151718191A1C1D1E1F21222324	D5AF94DE93487D1F3A8C577CB84A66A4	D5AF94DE93487D1F3A8C577CB84A66A4	Pass
DF769380D212792D026F049E2E3E48EF	262728292B2C2D2E30313233353637383A3B3C3D3F404142	84BDAC569CAE2828705F267CC8376E90	84BDAC569CAE2828705F267CC8376E90	Pass
79F374BC445BDABF8FCCB8843D6054C6	44454647494A4B4C4E4F50515354555658595A5B5D5E5F60	F7401DDA5AD5AB712B7EB5D10C6F99B6	F7401DDA5AD5AB712B7EB5D10C6F99B6	Pass
4E02F1242FA56B05C68DBAE8FE44C9D6	626364656768696A6C6D6E6F71727374767778797B7C7D7E	1C9D54318539EBD4C3B5B7E37BF119F0	1C9D54318539EBD4C3B5B7E37BF119F0	Pass

Test results for Fast Implementation (AES) - 256bit Key

Plaintext	Key	Expected Result	Actual Result	Pass?
834EADFCCAC7E1B30664B1ABA44815AB	000102030506070800A0B0C0D0F10111214151617191A1B1C1E1F202123242526	1946DABF6A03A2A2C3D0B05080AED6FC	1946DABF6A03A2A2C3D0B05080AED6FC	Pass
D9DC4DBA3021B05D67C0518F72B62BF1	28292A2B2D2E2F30323334353738393A3C3D3E3F41424344464748494B4C4D4E	5ED301D747D3CC715445EBDEC62F2FB4	5ED301D747D3CC715445EBDEC62F2FB4	Pass
A291D86301A4A739F7392173AA3C604C	50515253555657585A5B5C5D5F60616264656667696A6B6C6E6F707173747576	6585C8F43D13A6BEAB6419FC5935B9D0	6585C8F43D13A6BEAB6419FC5935B9D0	Pass
4264B2696498DE4DF79788A9F83E9390	78797A7B7D7E7F80828384858788898A8C8D8E8F91929394969798999B9C9D9E	2A5B56A596680FCC0E05F5E0F151ECAE	2A5B56A596680FCC0E05F5E0F151ECAE	Pass
EE9932B3721804D5A83EF5949245B6F6	A0A1A2A3A5A6A7A8AAABACADAFB0B1B2B4B5B6B7B9BABBBCBEBFC0C1C3C4C5C6	F5D6FF414FD2C6181494D20C37F2B8C4	F5D6FF414FD2C6181494D20C37F2B8C4	Pass
E6248F55C5FDCBCA9CBBB01C88A2EA77	C8C9CACBCDCECFD0D2D3D4D5D7D8D9DADCDDDEDFE1E2E3E4E6E7E8E9EBECEDEE	85399C01F59FFFB5204F19F8482F00B8	85399C01F59FFFB5204F19F8482F00B8	Pass
B8358E41B9DFF65FD461D55A99266247	F0F1F2F3F5F6F7F8FAFBFCFDFE01000204050607090A0B0C0E0F101113141516	92097B4C88A041DDF98144BC8D22E8E7	92097B4C88A041DDF98144BC8D22E8E7	Pass
F0E2D72260AF58E21E015AB3A4C0D906	18191A1B1D1E1F20222324252728292A2C2D2E2F31323334363738393B3C3D3E	89BD5B73B356AB412AEF9F76CEA2D65C	89BD5B73B356AB412AEF9F76CEA2D65C	Pass
475B8B823CE8893DB3C44A9F2A379FF7	40414243454647484A4B4C4D4F50515254555657595A5B5C5E5F606163646566	2536969093C55FF9454692F2FAC2F530	2536969093C55FF9454692F2FAC2F530	Pass
688F5281945812862F5F3076CF80412F	68696A6B6D6E6F70727374757778797A7C7D7E7F81828384868788898B8C8D8E	07FC76A872843F3F6E0081EE9396D637	07FC76A872843F3F6E0081EE9396D637	Pass
08D1D2BC750AF553365D35E75AFACEAA	90919293959697989A9B9C9D9FA0A1A2A4A5A6A7A9AAABACAEAFB0B1B3B4B5B6	E38BA8EC2AA741358DCC93E8F141C491	E38BA8EC2AA741358DCC93E8F141C491	Pass
8707121F47CC3EFCECA5F9A8474950A1	B8B9BABBBDBEBFC0C2C3C4C5C7C8C9CACCCDCECFD1D2D3D4D6D7D8D9DBDCDDDE	D028EE23E4A89075D0B03E868D7D3A42	D028EE23E4A89075D0B03E868D7D3A42	Pass
E51AA0B135DBA566939C3B6359A980C5	E0E1E2E3E5E6E7E8EAEBECEDEFF0F1F2F4F5F6F7F9FAFBFCFEFE010103040506	8CD9423DFC459E547155C5D1D522E540	8CD9423DFC459E547155C5D1D522E540	Pass
069A007FC76A459F98BAF917FEDF9521	08090A0B0D0E0F10121314151718191A1C1D1E1F21222324262728292B2C2D2E	080E9517EB1677719ACF728086040AE3	080E9517EB1677719ACF728086040AE3	Pass
726165C1723FBCF6C026D7D00B091027	30313233353637383A3B3C3D3F40414244454647494A4B4C4E4F505153545556	7C1700211A3991FC0ECDED0AB3E576B0	7C1700211A3991FC0ECDED0AB3E576B0	Pass
D7C544DE91D55CFCDE1F84CA382200CE	58595A5B5D5E5F60626364656768696A6C6D6E6F71727374767778797B7C7D7E	DABCBCC855839251DB51E224FBE87435	DABCBCC855839251DB51E224FBE87435	Pass
FED3C9A161B9B5B2BD611B41DC9DA357	80818283858687888A8B8C8D8F90919294959697999A9B9C9E9FA0A1A3A4A5A6	68D56FAD0406947A4DD27A7448C10F1D	68D56FAD0406947A4DD27A7448C10F1D	Pass
4F634CDC6551043409F30B635832CF82	A8A9AAABADAEAFB0B2B3B4B5B7B8B9BABCBDBEBFC1C2C3C4C6C7C8C9CBCCCDCE	DA9A11479844D1FFEE24BBF3719A9925	DA9A11479844D1FFEE24BBF3719A9925	Pass
109CE98DB0DFB36734D9F3394711B4E6	D0D1D2D3D5D6D7D8DADBDCDDDFE0E1E2E4E5E6E7E9EAEBECEEEFF0F1F3F4F5F6	5E4BA572F8D23E738DA9B05BA24B8D81	5E4BA572F8D23E738DA9B05BA24B8D81	Pass
4EA6DFABA2D8A02FFDFFA89835987242	70717273757677787A7B7C7D7F80818284858687898A8B8C8E8F909193949596	A115A2065D667E3F0B883837A6E903F8	A115A2065D667E3F0B883837A6E903F8	Pass
5AE094F54AF58E6E3CDBF976DAC6D9EF	98999A9B9D9E9FA0A2A3A4A5A7A8A9AAACADAEAFB1B2B3B4B6B7B8B9BBBCBDBE	3E9E90DC33EAC2437D86AD30B137E66E	3E9E90DC33EAC2437D86AD30B137E66E	Pass
764D8E8E0F29926DBE5122E66354FDBE	C0C1C2C3C5C6C7C8CACBCCCDCFD0D1D2D4D5D6D7D9DADBDCDEDFE0E1E3E4E5E6	01CE82D8FBCDAE824CB3C48E495C3692	01CE82D8FBCDAE824CB3C48E495C3692	Pass

3F0418F888CDF29A982BF6B75410D6A9	E8E9EAEBEDEEEFF0F2F3F4F5F7F8F9FAFCFDFEFF01020304060708090B0C0D0E	0C9CFF163CE936FAAF083CFD3DEA3117	0C9CFF163CE936FAAF083CFD3DEA3117	Pass
E4A3E7CB12CDD56AA4A75197A9530220	10111213151617181A1B1C1D1F20212224252627292A2B2C2E2F303133343536	5131BA9BD48F2BBA85560680DF504B52	5131BA9BD48F2BBA85560680DF504B52	Pass
211677684AAC1EC1A160F44C4EBF3F26	38393A3B3D3E3F40424344454748494A4C4D4E4F51525354565758595B5C5D5E	9DC503BBF09823AEC8A977A5AD26CCB2	9DC503BBF09823AEC8A977A5AD26CCB2	Pass
D21E439FF749AC8F18D6D4B105E03895	60616263656667686A6B6C6D6F70717274757677797A7B7C7E7F808183848586	9A6DB0C0862E506A9E397225884041D7	9A6DB0C0862E506A9E397225884041D7	Pass
D9F6FF44646C4725BD4C0103FF5552A7	88898A8B8D8E8F90929394959798999A9C9D9E9FA1A2A3A4A6A7A8A9ABACADAE	430BF9570804185E1AB6365FC6A6860C	430BF9570804185E1AB6365FC6A6860C	Pass
0B1256C2A00B976250CFC5B0C37ED382	B0B1B2B3B5B6B7B8BABBBCBDBFC0C1C2C4C5C6C7C9CACBCCCECFD0D1D3D4D5D6	3525EBC02F4886E6A5A3762813E8CE8A	3525EBC02F4886E6A5A3762813E8CE8A	Pass
B056447FFC6DC4523A36CC2E972A3A79	D8D9DADBDDDEDFE0E2E3E4E5E7E8E9EAECEDEEEFF1F2F3F4F6F7F8F9FBFCFDFE	07FA265C763779CCE224C7BAD671027B	07FA265C763779CCE224C7BAD671027B	Pass
5E25CA78F0DE55802524D38DA3FE4456	00010203050607080A0B0C0D0F10111214151617191A1B1C1E1F202123242526	E8B72B4E8BE243438C9FFF1F0E205872	E8B72B4E8BE243438C9FFF1F0E205872	Pass
A5BCF4728FA5EAAD8567C0DC24675F83	28292A2B2D2E2F30323334353738393A3C3D3E3F41424344464748494B4C4D4E	109D4F999A0E11ACE1F05E6B22CBCB50	109D4F999A0E11ACE1F05E6B22CBCB50	Pass
814E59F97ED84646B78B2CA022E9CA43	50515253555657585A5B5C5D5F60616264656667696A6B6C6E6F707173747576	45A5E8D4C3ED58403FF08D68A0CC4029	45A5E8D4C3ED58403FF08D68A0CC4029	Pass
15478BEEC58F4775C7A7F5D4395514D7	78797A7B7D7E7F80828384858788898A8C8D8E8F91929394969798999B9C9D9E	196865964DB3D417B6BD4D586BCB7634	196865964DB3D417B6BD4D586BCB7634	Pass
253548FFCA461C67C8CBC78CD59F4756	A0A1A2A3A5A6A7A8AAABACADAFB0B1B2B4B5B6B7B9BABBBCBEBFC0C1C3C4C5C6	60436AD45AC7D30D99195F815D98D2AE	60436AD45AC7D30D99195F815D98D2AE	Pass
FD7AD8D73B9B0F8CC41600640F503D65	C8C9CACBCDCECFD0D2D3D4D5D7D8D9DADCDDDEDFE1E2E3E4E6E7E8E9EBECEDEE	BB07A23F0B61014B197620C185E2CD75	BB07A23F0B61014B197620C185E2CD75	Pass
06199DE52C6CBF8AF954CD65830BCD56	F0F1F2F3F5F6F7F8FAFBFCFDFE01000204050607090A0B0C0E0F101113141516	5BC0B2850129C854423AFF0751FE343B	5BC0B2850129C854423AFF0751FE343B	Pass
F17C4FFE48E44C61BD891E257E725794	18191A1B1D1E1F20222324252728292A2C2D2E2F31323334363738393B3C3D3E	7541A78F96738E6417D2A24BD2BECA40	7541A78F96738E6417D2A24BD2BECA40	Pass
9A5B4A402A3E8A59BE6BF5CD8154F029	40414243454647484A4B4C4D4F50515254555657595A5B5C5E5F606163646566	B0A303054412882E464591F1546C5B9E	B0A303054412882E464591F1546C5B9E	Pass
79BD40B91A7E07DC939D441782AE6B17	68696A6B6D6E6F70727374757778797A7C7D7E7F81828384868788898B8C8D8E	778C06D8A355EEEE214FCEA14B4E0EEF	778C06D8A355EEEE214FCEA14B4E0EEF	Pass
D8CEAAF8976E5FBE1012D8C84F323799	90919293959697989A9B9C9D9FA0A1A2A4A5A6A7A9AAABACAEAFB0B1B3B4B5B6	09614206D15CBACE63227D06DB6BEEBB	09614206D15CBACE63227D06DB6BEEBB	Pass
3316E2751E2E388B083DA23DD6AC3FBE	B8B9BABBBDBEBFC0C2C3C4C5C7C8C9CACCCDCECFD1D2D3D4D6D7D8D9DBDCDDDE	41B97FB20E427A9FDBBB358D9262255D	41B97FB20E427A9FDBBB358D9262255D	Pass
8B7CFBE37DE7DCA793521819242C5816	E0E1E2E3E5E6E7E8EAEBECEDEFF0F1F2F4F5F6F7F9FAFBFCFEFE010103040506	C1940F703D845F957652C2D64ABD7ADF	C1940F703D845F957652C2D64ABD7ADF	Pass
F23F033C0EEBF8EC55752662FD58CE68	08090A0B0D0E0F10121314151718191A1C1D1E1F21222324262728292B2C2D2E	D2D44FCDAE5332343366DB297EFCF21B	D2D44FCDAE5332343366DB297EFCF21B	Pass
59EB34F6C8BDBACC5FC6AD73A59A1301	30313233353637383A3B3C3D3F40414244454647494A4B4C4E4F505153545556	EA8196B79DBE167B6AA9896E287EED2B	EA8196B79DBE167B6AA9896E287EED2B	Pass
DCDE8B6BD5CF7CC22D9505E3CE81261A	58595A5B5D5E5F60626364656768696A6C6D6E6F71727374767778797B7C7D7E	D6B0B0C4BA6C7DBE5ED467A1E3F06C2D	D6B0B0C4BA6C7DBE5ED467A1E3F06C2D	Pass
E33CF7E524FED781E7042FF9F4B35DC7	80818283858687888A8B8C8D8F90919294959697999A9B9C9E9FA0A1A3A4A5A6	EC51EB295250C22C2FB01816FB72BCAE	EC51EB295250C22C2FB01816FB72BCAE	Pass
27963C8FACDF73062867D164DF6D064C	A8A9AAABADAEAFB0B2B3B4B5B7B8B9BABCBDBEBFC1C2C3C4C6C7C8C9CBCCCDCE	ADED6630A07CE9C7408A155D3BD0D36F	ADED6630A07CE9C7408A155D3BD0D36F	Pass
77B1CE386B551B995F2F2A1DA994EEF8	D0D1D2D3D5D6D7D8DADBDCDDDFE0E1E2E4E5E6E7E9EAEBECEEEFF0F1F3F4F5F6	697C9245B9937F32F5D1C82319F0363A	697C9245B9937F32F5D1C82319F0363A	Pass
F083388B013679EFCF0BB9B15D52AE5C	F8F9FAFBFDFEFE00020304050708090A0C0D0E0F11121314161718191B1C1D1E	AAD5AD50C6262AAEC30541A1B7B5B19C	AAD5AD50C6262AAEC30541A1B7B5B19C	Pass
C5009E0DAB55DB0ABDB636F2600290C8	20212223252627282A2B2C2D2F30313234353637393A3B3C3E3F404143444546	7D34B893855341EC625BD6875AC18C0D	7D34B893855341EC625BD6875AC18C0D	Pass

Test results for Fast Implementation (QAES) - 128bit Key

Plaintext	Key	Expected Output	Actual Output	Pass?
506812A45F08C889B97F5980038B8359	00010203050607080A0B0C0D0F101112	D8F532538289EF7D06B506A4FD5BE9C9	D8F532538289EF7D06B506A4FD5BE9C9	Pass
5C6D71CA30DE8B8B00549984D2EC7D4B	14151617191A1B1C1E1F202123242526	59AB30F4D4EE6E4FF9907EF65B1FB68C	59AB30F4D4EE6E4FF9907EF65B1FB68C	Pass
53F3F4C64F8616E4E7C56199F48F21F6	28292A2B2D2E2F30323334353738393A	BF1ED2FCB2AF3FD41443B56D85025CB1	BF1ED2FCB2AF3FD41443B56D85025CB1	Pass
A1EB65A3487165FB0F1C27FF9959F703	3C3D3E3F41424344464748494B4C4D4E	7316632D5C32233EDCB0780560EAE8B2	7316632D5C32233EDCB0780560EAE8B2	Pass
3553ECF0B1739558B08E350A98A39BFA	50515253555657585A5B5C5D5F606162	408C073E3E2538072B72625E68B8364B	408C073E3E2538072B72625E68B8364B	Pass
67429969490B9711AE2B01DC497AFDE8	64656667696A6B6C6E6F707173747576	E1F94DFA776597BEACA262F2F6366FEA	E1F94DFA776597BEACA262F2F6366FEA	Pass
93385C1F2AEC8BED192F5A8E161DD508	78797A7B7D7E7F80828384858788898A	F29E986C6A1C27D7B29FFD7EE92B75F1	F29E986C6A1C27D7B29FFD7EE92B75F1	Pass
B5BF946BE19BEB8DB3983B5F4C6E8DDB	8C8D8E8F91929394969798999B9C9D9E	131C886A57F8C2E713ABA6955E2B55B5	131C886A57F8C2E713ABA6955E2B55B5	Pass
41321EE10E21BD907227C4450FF42324	A0A1A2A3A5A6A7A8AAABACADAFB0B1B2	D2AB7662DF9B8C740210E5EEB61C199D	D2AB7662DF9B8C740210E5EEB61C199D	Pass
00A82F59C91C8486D12C0A80124F6089	B4B5B6B7B9BABBBCBEBFC0C1C3C4C5C6	14C10554B2859C484CAB5869BBE7C470	14C10554B2859C484CAB5869BBE7C470	Pass
7CE0FD076754691B4BBD9FAF8A1372FE	C8C9CACBCDCECFD0D2D3D4D5D7D8D9DA	DB4D498F0A49CF55445D502C1F9AB3B5	DB4D498F0A49CF55445D502C1F9AB3B5	Pass
23605A8243D07764541BC5AD355B3129	DCDDDEDFE1E2E3E4E6E7E8E9EBECEDEE	6D96FEF7D66590A77A77BB2056667F7F	6D96FEF7D66590A77A77BB2056667F7F	Pass
12A8CFA23EA764FD876232B4E842BC44	F0F1F2F3F5F6F7F8FAFBFCFDFE010002	316FB68EDBA736C53E78477BF913725C	316FB68EDBA736C53E78477BF913725C	Pass
BCAF32415E8308B3723E5FDD853CCC80	04050607090A0B0C0E0F101113141516	6936F2B93AF8397FD3A771FC011C8C37	6936F2B93AF8397FD3A771FC011C8C37	Pass
89AFAE685D801AD747ACE91FC49ADDE0	2C2D2E2F31323334363738393B3C3D3E	F3F92F7A9C59179C1FCC2C2BA0B082CD	F3F92F7A9C59179C1FCC2C2BA0B082CD	Pass
F521D07B484357C4A69E76124A634216	40414243454647484A4B4C4D4F505152	6A95EA659EE3889158E7A9152FF04EBC	6A95EA659EE3889158E7A9152FF04EBC	Pass
3E23B3BC065BCC152407E23896D77783	54555657595A5B5C5E5F606163646566	1959338344E945670678A5D432C90B93	1959338344E945670678A5D432C90B93	Pass
79F0FBA002BE1744670E7E99290D8F52	68696A6B6D6E6F70727374757778797A	E49BDDD2369B83EE66E6C75A1161B394	E49BDDD2369B83EE66E6C75A1161B394	Pass
DA23FE9D5BD63E1D72E3DAFBE21A6C2A	7C7D7E7F81828384868788898B8C8D8E	D3388F19057FF704B70784164A74867D	D3388F19057FF704B70784164A74867D	Pass
E3F5698BA90B6A022EFD7DB2C7E6C823	A4A5A6A7A9AAABACAEAFB0B1B3B4B5B6	23AA03E2D5E4CD24F3217E596480D1E1	23AA03E2D5E4CD24F3217E596480D1E1	Pass
BDC2691D4F1B73D2700679C3BCBF9C6E	E0E1E2E3E5E6E7E8EAEBECEDEFF0F1F2	C84113D68B666AB2A50A8BDB222E91B9	C84113D68B666AB2A50A8BDB222E91B9	Pass
BA74E02093217EE1BA1B42BD5624349A	08090A0B0D0E0F10121314151718191A	AC02403981CD4340B507963DB65CB7B6	AC02403981CD4340B507963DB65CB7B6	Pass

B5C593B5851C57FBF8B3F57715E8F680	6C6D6E6F717273747677787 97B7C7D7E	8D1299236223359474011F6BF5088414	8D1299236223359474011F6BF5088414	Pass
3DA9BD9CEC072381788F9387C3BBF4EE	808182838586878888A8B8C8D8F909192	5A1D6AB8605505F7977E55B9A54D9B90	5A1D6AB8605505F7977E55B9A54D9B90	Pass
4197F3051121702AB65D316B3C637374	94959697999A9B9C9E9FA0A1A3A4A5A6	72E9C2D519CF555E4208805AABE3B258	72E9C2D519CF555E4208805AABE3B258	Pass
9F46C62EC4F6EE3F6E8C62554BC48AB7	A8A9AAABADAEAFB0B2B3B4B5B7B8B9BA	A8F3E81C4A23A39EF4D745DFFE026E80	A8F3E81C4A23A39EF4D745DFFE026E80	Pass
0220673FE9E699A4EBC8E0DBEB6979C8	BCBDBEBFC1C2C3C4C6C7C8C9CBCCCDCE	546F646449D31458F9EB4EF5483AEE6C	546F646449D31458F9EB4EF5483AEE6C	Pass
B2B99171337DED9BC8C2C23FF6F18867	D0D1D2D3D5D6D7D8DADBDCDDDFE0E1E2	4DBE4BC84AC797C0EE4EFB7F1A07401C	4DBE4BC84AC797C0EE4EFB7F1A07401C	Pass
A7FACF4E301E984E5EFEEFD645B23505	E4E5E6E7E9EAEBECEEEFF0F1F3F4F5F6	25E10BFB411BBD4D625AC8795C8CA3B3	25E10BFB411BBD4D625AC8795C8CA3B3	Pass
F7C762E4A9819160FD7ACFB6C4EEDCDD	F8F9FAFBFDFEFE00020304050708090A	315637405054EC803614E43DEF177579	315637405054EC803614E43DEF177579	Pass
9B64FC21EA08709F4915436FAA70F1BE	0C0D0E0F11121314161718191B1C1D1E	60C5BC8A1410247295C6386C59E572A8	60C5BC8A1410247295C6386C59E572A8	Pass
52AF2C3DE07EE6777F55A4ABFC100B3F	20212223252627282A2B2C2D2F303132	01366FC8CA52DFE055D6A00A76471BA6	01366FC8CA52DFE055D6A00A76471BA6	Pass
2FCA001224386C57AA3F968CBE2C816F	34353637393A3B3C3E3F404143444546	ECC46595516EC612449C3F581E7D42FF	ECC46595516EC612449C3F581E7D42FF	Pass
4149C73658A4A9C564342755EE2C132F	48494A4B4D4E4F50525354555758595A	6B7FFE4C602A154B06EE9C7DAB5331C9	6B7FFE4C602A154B06EE9C7DAB5331C9	Pass
AF60005A00A1772F7C07A48A923C23D2	5C5D5E5F61626364666768696B6C6D6E	7DA234C14039A240DD02DD0FBF84EB67	7DA234C14039A240DD02DD0FBF84EB67	Pass
6FCCBC28363759914B6F0280AFAF20C6	70717273757677787A7B7C7D7F808182	C7DC217D9E3604FFE7E91F080ECD5A3A	C7DC217D9E3604FFE7E91F080ECD5A3A	Pass
7D82A43DDF4FEFA2FC5947499884D386	84858687898A8B8C8E8F909193949596	37785901863F5C81260EA41E7580CDA5	37785901863F5C81260EA41E7580CDA5	Pass
5D5A990EAAB9093AFE4CE254DFA49EF9	98999A9B9D9E9FA0A2A3A4A5A7A8A9AA	A07B9338E92ED105E6AD720FCCCE9FE4	A07B9338E92ED105E6AD720FCCCE9FE4	Pass
4CD1E2FD3F4434B553AAE453F0ED1A02	ACADAEAFB1B2B3B4B6B7B8B9BBBCBDBE	AE0FB9722418CC21A7DA816BBC61322C	AE0FB9722418CC21A7DA816BBC61322C	Pass
5A2C9A9641D4299125FA1B9363104B5E	C0C1C2C3C5C6C7C8CACBCCCDCFD0D1D2	C826A193080FF91FFB21F71D3373C877	C826A193080FF91FFB21F71D3373C877	Pass
B517FE34C0FA217D341740BFD4FE8DD4	D4D5D6D7D9DADBDCDEDFE0E1E3E4E5E6	1181B11B0E494E8D8B0AA6B1D5AC2C48	1181B11B0E494E8D8B0AA6B1D5AC2C48	Pass
014BAF2278A69D331D5180103643E99A	E8E9EAEBEDEEEFF0F2F3F4F5F7F8F9FA	6743C3D1519AB4F2CD9A78AB09A511BD	6743C3D1519AB4F2CD9A78AB09A511BD	Pass
B529BD8164F20D0AA443D4932116841C	FCFDFEFF01020304060708090B0C0D0E	DC55C076D52BACDF2EEFD952946A439D	DC55C076D52BACDF2EEFD952946A439D	Pass
2E596DCBB2F33D4216A1176D5BD1E456	10111213151617181A1B1C1D1F202122	711B17B590FFC72B5C8E342B601E8003	711B17B590FFC72B5C8E342B601E8003	Pass
7274A1EA2B7EE2424E9A0E4673689143	24252627292A2B2C2E2F303133343536	19983BB0950783A537E1339F4AA21C75	19983BB0950783A537E1339F4AA21C75	Pass
AE20020BD4F13E9D90140BEE3B5D26AF	38393A3B3D3E3F40424344454748494A	3BA7762E15554169C0F4FA39164C410C	3BA7762E15554169C0F4FA39164C410C	Pass
BAAC065DA7AC26E855E79C8849D75A02	4C4D4E4F51525354565758595B5C5D5E	A0564C41245AFCA7AF8AA2E0E588EA89	A0564C41245AFCA7AF8AA2E0E588EA89	Pass
7C917D8D1D45FAB9E2540E28832540CC	60616263656667686A6B6C6D6F707172	5E36A42A2E099F54AE85ECD92E2381ED	5E36A42A2E099F54AE85ECD92E2381ED	Pass
BDE6F89E16DAADB0E847A2A614566A91	74757677797A7B7C7E7F808183848586	770036F878CD0F6CA2268172F106F2FE	770036F878CD0F6CA2268172F106F2FE	Pass
C9DE163725F1F5BE44EBB1DB51D07FBC	88898A8B8D8E8F90929394959798999A	7E4E03908B716116443CCF7C94E7C259	7E4E03908B716116443CCF7C94E7C259	Pass

Test results for Fast Implementation (QAES) - 192bit Key

Plaintext	Key	Expected Result	Actual Result	Pass?
2D33EEF2C0430A8A9EBF45E809C40BB6	00010203050607080A0B0C0D0F10111214151617191A1B1C	DFF4945E0336DF4C1C56BC700EFF837F	DFF4945E0336DF4C1C56BC700EFF837F	Pass
6AA375D1FA155A61FB72353E0A5A8756	1E1F20212324252628292A2B2D2E2F30323334353738393A	B6FDDEF4752765E347D5D2DC196D1252	B6FDDEF4752765E347D5D2DC196D1252	Pass
BC3736518B9490DCB8ED60EB26758ED4	3C3D3E3F41424344464748494B4C4D4E5051525355565758	D23684E3D963B3AFCF1A114ACA90CBD6	D23684E3D963B3AFCF1A114ACA90CBD6	Pass
AA214402B46CFFB9F761EC11263A311E	5A5B5C5D5F60616264656667696A6B6C6E6F707173747576	3A7AC027753E2A18C2CEAB9E17C11FD0	3A7AC027753E2A18C2CEAB9E17C11FD0	Pass
02AEA86E572EEAB66B2C3AF5E9A46FD6	78797A7B7D7E7F80828384858788898A8C8D8E8F91929394	8F6786BD007528BA26603C1601CDD0D8	8F6786BD007528BA26603C1601CDD0D8	Pass
E2AEF6ACC33B965C4FA1F91C75FF6F36	969798999B9C9D9EA0A1A2A3A5A6A7A8AAABACADAFB0B1B2	D17D073B01E71502E28B47AB551168B3	D17D073B01E71502E28B47AB551168B3	Pass
0659DF46427162B9434865DD9499F91D	B4B5B6B7B9BABBBCBEBFC0C1C3C4C5C6C8C9CACBCDCECFD0	A469DA517119FAB95876F41D06D40FFA	A469DA517119FAB95876F41D06D40FFA	Pass
49A44239C748FEB456F59C276A5658DF	D2D3D4D5D7D8D9DADCDDDEDFE1E2E3E4E6E7E8E9EBECEDEE	6091AA3B695C11F5C0B6AD26D3D862FF	6091AA3B695C11F5C0B6AD26D3D862FF	Pass
66208F6E9D04525BDEDB2733B6A6BE37	F0F1F2F3F5F6F7F8FAFBFCFDFE01000204050607090A0B0C	70F9E67F9F8DF1294131662DC6E69364	70F9E67F9F8DF1294131662DC6E69364	Pass
3393F8DFC729C97F5480B950BC9666B0	0E0F10111314151618191A1B1D1E1F20222324252728292A	D154DCAFAD8B207FA5CBC95E9996B559	D154DCAFAD8B207FA5CBC95E9996B559	Pass
606834C8CE063F3234CF1145325DBD71	2C2D2E2F31323334363738393B3C3D3E4041424345464748	4934D541E8B46FA339C805A7AEB9E5DA	4934D541E8B46FA339C805A7AEB9E5DA	Pass
FEC1C04F529BBD17D8CECFCC4718B17F	4A4B4C4D4F50515254555657595A5B5C5E5F606163646566	62564C738F3EFE186E1A127A0C4D3C61	62564C738F3EFE186E1A127A0C4D3C61	Pass
32DF99B431ED5DC5ACF8CAF6DC6CE475	68696A6B6D6E6F70727374757778797A7C7D7E7F81828384	07805AA043986EB23693E23BEF8F3438	07805AA043986EB23693E23BEF8F3438	Pass
7FDC2B746F3F665296943B83710D1F82	868788898B8C8D8E90919293959697989A9B9C9D9FA0A1A2	DF0B4931038BADE848DEE3B4B85AA44B	DF0B4931038BADE848DEE3B4B85AA44B	Pass
8FBA1510A3C5B87E2EAA3F7A91455CA2	A4A5A6A7A9AAABACAEAFB0B1B3B4B5B6B8B9BABBBDBEBFC0	592D5FDED76582E4143C65099309477C	592D5FDED76582E4143C65099309477C	Pass
2C9B468B1C2EED92578D41B0716B223B	C2C3C4C5C7C8C9CACCCDCECFD1D2D3D4D6D7D8D9DBDCDDDE	C9B8D6545580D3DFBCDD09B954ED4E92	C9B8D6545580D3DFBCDD09B954ED4E92	Pass
0A2BBF0EFC6BC0034F8A03433FCA1B1A	E0E1E2E3E5E6E7E8EAEBECEDEFF0F1F2F4F5F6F7F9FAFBFC	5DCCD5D6EB7C1B42ACB008201DF707A0	5DCCD5D6EB7C1B42ACB008201DF707A0	Pass
25260E1F31F4104D387222E70632504B	FEFE01010304050608090A0B0D0E0F10121314151718191A	A2A91682FFEB6ED1D34340946829E6F9	A2A91682FFEB6ED1D34340946829E6F9	Pass
C527D25A49F08A5228D338642AE65137	1C1D1E1F21222324262728292B2C2D2E3031323335363738	E45D185B797000348D9267960A68435D	E45D185B797000348D9267960A68435D	Pass
3B49FC081432F5890D0E3D87E884A69E	3A3B3C3D3F40414244454647494A4B4C4E4F505153545556	45E060DAE5901CDA8089E10D4F4C246B	45E060DAE5901CDA8089E10D4F4C246B	Pass
D173F9ED1E57597E166931DF2754A083	58595A5B5D5E5F60626364656768696A6C6D6E6F71727374	F6951AFACC0079A369C71FDCFF45DF50	F6951AFACC0079A369C71FDCFF45DF50	Pass
8C2B7CAFA5AFE7F13562DAEAE1ADEDE0	767778797B7C7D7E80818283858687888A8B8C8D8F909192	9E95E00F351D5B3AC3D0E22E626DDAD6	9E95E00F351D5B3AC3D0E22E626DDAD6	Pass
AAF4EC8C1A815AEB826CAB741339532C	94959697999A9B9C9E9FA0A1A3A4A5A6A8A9AAABADAEAFB0	9CB566FF26D92DAD083B51FDC18C173C	9CB566FF26D92DAD083B51FDC18C173C	Pass

40BE8C5D9108E663F38F1A2395279ECF	D0D1D2D3D5D6D7D8DADBDCDDDFE0E1E2E4E5E6E7E9EAEBEC	C9C82766176A9B228EB9A974A010B4FB	C9C82766176A9B228EB9A974A010B4FB	Pass
0C8AD9BC32D43E04716753AA4CFBE351	2A2B2C2D2F30313234353637393A3B3C3E3F404143444546	D8E26AA02945881D5137F1C1E1386E88	D8E26AA02945881D5137F1C1E1386E88	Pass
1407B1D5F87D63357C8DC7EBBAEBBFEE	48494A4B4D4E4F5052535455575859 5A5C5D5E5F61626364	C0E024CCD68FF5FFA4D139C355A77C55	C0E024CCD68FF5FFA4D139C355A77C55	Pass
E62734D1AE3378C4549E939E6F123416	84858687898A8B8C8E8F90919394959698999A9B9D9E9FA0	0B18B3D16F491619DA338640DF391D43	0B18B3D16F491619DA338640DF391D43	Pass
5A752CFF2A176DB1A1DE77F2D2CDEE41	A2A3A4A5A7A8A9AAACADAEAFB1B2B3B4B6B7B8B9BBBCBDBE	DBE09AC8F66027BF20CB6E434F252EFC	DBE09AC8F66027BF20CB6E434F252EFC	Pass
A9C8C3A4EABEDC80C64730DDD018CD88	C0C1C2C3C5C6C7C8CACBCCCDCFD0D1D2D4D5D6D7D9DADBDC	6D04E5E43C5B9CBE05FEB9606B6480FE	6D04E5E43C5B9CBE05FEB9606B6480FE	Pass
EE9B3DBBDB86180072130834D305999A	1A1B1C1D1F20212224252627292A2B2C2E2F303133343536	DD1D6553B96BE526D9FEE0FBD7176866	DD1D6553B96BE526D9FEE0FBD7176866	Pass
A7FA8C3586B8EBDE7568EAD6F634A879	38393A3B3D3E3F40424344454748494A4C4D4E4F51525354	0260CA7E3F979FD015B0DD4690E16D2A	0260CA7E3F979FD015B0DD4690E16D2A	Pass
37E0F4A87F127D45AC936FE7AD88C10A	929394959798999A9C9D9E9FA1A2A3A4A6A7A8A9ABACADAE	9893734DE10EDCC8A67C3B110B8B8CC6	9893734DE10EDCC8A67C3B110B8B8CC6	Pass
3F77D8B5D92BAC148E4E46F697A535C5	464748494B4C4D4E50515253555657585A5B5C5D5F606162	93B30B750516B2D18808D710C2EE84EF	93B30B750516B2D18808D710C2EE84EF	Pass
D25EBB686C40F7E2C4DA1014936571CA	828384858788898A8C8D8E8F91929394969798999B9C9D9E	16F65FA47BE3CB5E6DFE7C6C37016C0E	16F65FA47BE3CB5E6DFE7C6C37016C0E	Pass
4F1C769D1E5B0552C7ECA84DEA26A549	A0A1A2A3A5A6A7A8AAABACADAFB0B1B2B4B5B6B7B9BABBBC	F3847210D5391E2360608E5ACB560581	F3847210D5391E2360608E5ACB560581	Pass
8548E2F882D7584D0FAFC54372B6633A	BEBFC0C1C3C4C5C6C8C9CACBCDCECFD0D2D3D4D5D7D8D9DA	8754462CD223366D0753913E6AF2643D	8754462CD223366D0753913E6AF2643D	Pass
87D7A336CB476F177CD2A51AF2A62CDF	DCDDDEDFE1E2E3E4E6E7E8E9EBECEDEEF0F1F2F3F5F6F7F8	1EA20617468D1B806A1FD58145462017	1EA20617468D1B806A1FD58145462017	Pass
03B1FEAC668C4E485C1065DFC22B44EE	FAFBFCFDFE01000204050607090A0B0C0E0F101113141516	3B155D927355D737C6BE9DDA60136E2E	3B155D927355D737C6BE9DDA60136E2E	Pass
BDA15E66819FA72D653A6866AA287962	18191A1B1D1E1F20222324252728292A2C2D2E2F31323334	26144F7B66DAA91B6333DBD3850502B3	26144F7B66DAA91B6333DBD3850502B3	Pass
4D0C7A0D2505B80BF8B62CEB12467F0A	363738393B3C3D3E40414243454647484A4B4C4D4F505152	E4F9A4AB52CED8134C649BF319EBCC90	E4F9A4AB52CED8134C649BF319EBCC90	Pass
626D34C9429B37211330986466B94E5F	54555657595A5B5C5E5F60616364656668696A6B6D6E6F70	B9DDD29AC6128A6CAB121E34A4C62B36	B9DDD29AC6128A6CAB121E34A4C62B36	Pass
333C3E6BF00656B088A17E5FF0E7F60A	727374757778797A7C7D7E7F81828384868788898B8C8D8E	6FCDDAD898F2CE4EFF51294F5EAAF5C9	6FCDDAD898F2CE4EFF51294F5EAAF5C9	Pass
687ED0CDC0D2A2BC8C466D05EF9D2891	90919293959697989A9B9C9D9FA0A1A2A4A5A6A7A9AAABAC	C9A6FE2BF4028080BEA6F7FC417BD7E3	C9A6FE2BF4028080BEA6F7FC417BD7E3	Pass
487830E78CC56C1693E64B2A6660C7B6	AEAFB0B1B3B4B5B6B8B9BABBBDBEBFC0C2C3C4C5C7C8C9CA	6A2026846D8609D60F298A9C0673127F	6A2026846D8609D60F298A9C0673127F	Pass
7A48D6B7B52B29392AA2072A32B66160	CCCDCECFD1D2D3D4D6D7D8D9DBDCDDDEE0E1E2E3E5E6E7E8	2CB25C005E26EFEA44336C4C97A4240B	2CB25C005E26EFEA44336C4C97A4240B	Pass
907320E64C8C5314D10F8D7A11C8618D	EAEBECEDEFF0F1F2F4F5F6F7F9FAFBFCFEFE010103040506	496967AB8680DDD73D09A0E4C7DCC8AA	496967AB8680DDD73D09A0E4C7DCC8AA	Pass
B561F2CA2D6E65A4A98341F3ED9FF533	08090A0B0D0E0F10121314151718191A1C1D1E1F21222324	D5AF94DE93487D1F3A8C577CB84A66A4	D5AF94DE93487D1F3A8C577CB84A66A4	Pass
DF769380D212792D026F049E2E3E48EF	262728292B2C2D2E30313233353637383A3B3C3D3F404142	84BDAC569CAE2828705F267CC8376E90	84BDAC569CAE2828705F267CC8376E90	Pass
79F374BC445BDABF8FCCB8843D6054C6	44454647494A4B4C4E4F50515354555658595A5B5D5E5F60	F7401DDA5AD5AB712B7EB5D10C6F99B6	F7401DDA5AD5AB712B7EB5D10C6F99B6	Pass
4E02F1242FA56B05C68DBAE8FE44C9D6	626364656768696A6C6D6E6F71727374767778797B7C7D7E	1C9D54318539EBD4C3B5B7E37BF119F0	1C9D54318539EBD4C3B5B7E37BF119F0	Pass

Test results for Fast Implementation (QAES) - 256bit Key

Plaintext	Key	Expected Result	Actual Result	Pass?
834EADFCCAC7E1B30664B1ABA44815AB	00010203050607080A0B0C0D0F10111214151617191A1B1C1E1F202123242526	1946DABF6A03A2A2C3D0B05080AED6FC	1946DABF6A03A2A2C3D0B05080AED6FC	Pass
D9DC4DBA3021B05D67C0518F72B62BF1	28292A2B2D2E2F30323334353738393A3C3D3E3F41424344464748494B4C4D4E	5ED301D747D3CC715445EBDEC62F2FB4	5ED301D747D3CC715445EBDEC62F2FB4	Pass
A291D86301A4A739F7392173AA3C604C	50515253555657585A5B5C5D5F60616264656667696A6B6C6E6F707173747576	6585C8F43D13A6BEAB6419FC5935B9D0	6585C8F43D13A6BEAB6419FC5935B9D0	Pass
4264B2696498DE4DF79788A9F83E9390	78797A7B7D7E7F80828384858788898A8C8D8E8F91929394969798999B9C9D9E	2A5B56A596680FCC0E05F5E0F151ECAE	2A5B56A596680FCC0E05F5E0F151ECAE	Pass
EE9932B3721804D5A83EF5949245B6F6	A0A1A2A3A5A6A7A8AAABACADAFB0B1B2B4B5B6B7B9BABBBCBEBFC0C1C3C4C5C6	F5D6FF414FD2C6181494D20C37F2B8C4	F5D6FF414FD2C6181494D20C37F2B8C4	Pass
E6248F55C5FDCBCA9CBBB01C88A2EA77	C8C9CACBCDCECFD0D2D3D4D5D7D8D9DADCDDDEDFE1E2E3E4E6E7E8E9EBECEDEE	85399C01F59FFFB5204F19F8482F00B8	85399C01F59FFFB5204F19F8482F00B8	Pass
B8358E41B9DFF65FD461D55A99266247	F0F1F2F3F5F6F7F8FAFBFCFDFE01000204050607090A0B0C0E0F101113141516	92097B4C88A041DDF98144BC8D22E8E7	92097B4C88A041DDF98144BC8D22E8E7	Pass
F0E2D72260AF58E21E015AB3A4C0D906	18191A1B1D1E1F20222324252728292A2C2D2E2F31323334363738393B3C3D3E	89BD5B73B356AB412AEF9F76CEA2D65C	89BD5B73B356AB412AEF9F76CEA2D65C	Pass
475B8B823CE8893DB3C44A9F2A379FF7	40414243454647484A4B4C4D4F50515254555657595A5B5C5E5F606163646566	2536969093C55FF9454692F2FAC2F530	2536969093C55FF9454692F2FAC2F530	Pass
688F5281945812862F5F3076CF80412F	68696A6B6D6E6F70727374757778797A7C7D7E7F81828384868788898B8C8D8E	07FC76A872843F3F6E0081EE9396D637	07FC76A872843F3F6E0081EE9396D637	Pass
08D1D2BC750AF553365D35E75AFACEAA	90919293959697989A9B9C9D9FA0A1A2A4A5A6A7A9AAABACAEAFB0B1B3B4B5B6	E38BA8EC2AA741358DCC93E8F141C491	E38BA8EC2AA741358DCC93E8F141C491	Pass
8707121F47CC3EFCECA5F9A8474950A1	B8B9BABBBDBEBFC0C2C3C4C5C7C8C9CACCCDCECFD1D2D3D4D6D7D8D9DBDCDDDE	D028EE23E4A89075D0B03E868D7D3A42	D028EE23E4A89075D0B03E868D7D3A42	Pass
E51AA0B135DBA566939C3B6359A980C5	E0E1E2E3E5E6E7E8EAEBECEDEFF0F1F2F4F5F6F7F9FAFBFCFEFE010103040506	8CD9423DFC459E547155C5D1D522E540	8CD9423DFC459E547155C5D1D522E540	Pass
069A007FC76A459F98BAF917FEDF9521	08090A0B0D0E0F10121314151718191A1C1D1E1F21222324262728292B2C2D2E	080E9517EB1677719ACF728086040AE3	080E9517EB1677719ACF728086040AE3	Pass
726165C1723FBCF6C026D7D00B091027	30313233353637383A3B3C3D3F40414244454647494A4B4C4E4F505153545556	7C1700211A3991FC0ECDED0AB3E576B0	7C1700211A3991FC0ECDED0AB3E576B0	Pass
D7C544DE91D55CFCDE1F84CA382200CE	58595A5B5D5E5F60626364656768696A6C6D6E6F71727374767778797B7C7D7E	DABCBCC855839251DB51E224FBE87435	DABCBCC855839251DB51E224FBE87435	Pass
FED3C9A161B9B5B2BD611B41DC9DA357	80818283858687888A8B8C8D8F90919294959697999A9B9C9E9FA0A1A3A4A5A6	68D56FAD0406947A4DD27A7448C10F1D	68D56FAD0406947A4DD27A7448C10F1D	Pass
4F634CDC6551043409F30B635832CF82	A8A9AAABADAEAFB0B2B3B4B5B7B8B9BABCBDBEBFC1C2C3C4C6C7C8C9CBCCCDCE	DA9A11479844D1FFEE24BBF3719A9925	DA9A11479844D1FFEE24BBF3719A9925	Pass
109CE98DB0DFB36734D9F3394711B4E6	D0D1D2D3D5D6D7D8DADBDCDDDFE0E1E2E4E5E6E7E9EAEBECEEEFF0F1F3F4F5F6	5E4BA572F8D23E738DA9B05BA24B8D81	5E4BA572F8D23E738DA9B05BA24B8D81	Pass
4EA6DFABA2D8A02FFDFFA89835987242	70717273757677787A7B7C7D7F80818284858687898A8B8C8E8F909193949596	A115A2065D667E3F0B883837A6E903F8	A115A2065D667E3F0B883837A6E903F8	Pass
5AE094F54AF58E6E3CDBF976DAC6D9EF	98999A9B9D9E9FA0A2A3A4A5A7A8A9AAACADAEAFB1B2B3B4B6B7B8B9BBBCBDBE	3E9E90DC33EAC2437D86AD30B137E66E	3E9E90DC33EAC2437D86AD30B137E66E	Pass
764D8E8E0F29926DBE5122E66354FDBE	C0C1C2C3C5C6C7C8CACBCCCDCFD0D1D2D4D5D6D7D9DADBDCDEDFE0E1E3E4E5E6	01CE82D8FBCDAE824CB3C48E495C3692	01CE82D8FBCDAE824CB3C48E495C3692	Pass

3F0418F888CDF29A982BF6B75410D6A9	E8E9EAEBEDEEEFF0F2F3F4F5F7F8F9FAFCFDFEFF01020304060708090B0C0D0E	0C9CFF163CE936FAAF083CFD3DEA3117	0C9CFF163CE936FAAF083CFD3DEA3117	Pass
E4A3E7CB12CDD56AA4A75197A9530220	10111213151617181A1B1C1D1F20212224252627292A2B2C2E2F303133343536	5131BA9BD48F2BBA85560680DF504B52	5131BA9BD48F2BBA85560680DF504B52	Pass
211677684AAC1EC1A160F44C4EBF3F26	38393A3B3D3E3F40424344454748494A4C4D4E4F51525354565758595B5C5D5E	9DC503BBF09823AEC8A977A5AD26CCB2	9DC503BBF09823AEC8A977A5AD26CCB2	Pass
D21E439FF749AC8F18D6D4B105E03895	60616263656667686A6B6C6D6F70717274757677797A7B7C7E7F808183848586	9A6DB0C0862E506A9E397225884041D7	9A6DB0C0862E506A9E397225884041D7	Pass
D9F6FF44646C4725BD4C0103FF5552A7	88898A8B8D8E8F90929394959798999A9C9D9E9FA1A2A3A4A6A7A8A9ABACADAE	430BF9570804185E1AB6365FC6A6860C	430BF9570804185E1AB6365FC6A6860C	Pass
0B1256C2A00B976250CFC5B0C37ED382	B0B1B2B3B5B6B7B8BABBBCBDBFC0C1C2C4C5C6C7C9CACBCCCECFD0D1D3D4D5D6	3525EBC02F4886E6A5A3762813E8CE8A	3525EBC02F4886E6A5A3762813E8CE8A	Pass
B056447FFC6DC4523A36CC2E972A3A79	D8D9DADBDDDEDFE0E2E3E4E5E7E8E9EAECEDEEEFF1F2F3F4F6F7F8F9FBFCFDFE	07FA265C763779CCE224C7BAD671027B	07FA265C763779CCE224C7BAD671027B	Pass
5E25CA78F0DE55802524D38DA3FE4456	00010203050607080A0B0C0D0F10111214151617191A1B1C1E1F202123242526	E8B72B4E8BE243438C9FFF1F0E205872	E8B72B4E8BE243438C9FFF1F0E205872	Pass
A5BCF4728FA5EAAD8567C0DC24675F83	28292A2B2D2E2F30323334353738393A3C3D3E3F41424344464748494B4C4D4E	109D4F999A0E11ACE1F05E6B22CBCB50	109D4F999A0E11ACE1F05E6B22CBCB50	Pass
814E59F97ED84646B78B2CA022E9CA43	50515253555657585A5B5C5D5F60616264656667696A6B6C6E6F707173747576	45A5E8D4C3ED58403FF08D68A0CC4029	45A5E8D4C3ED58403FF08D68A0CC4029	Pass
15478BEEC58F4775C7A7F5D4395514D7	78797A7B7D7E7F80828384858788898A8C8D8E8F91929394969798999B9C9D9E	196865964DB3D417B6BD4D586BCB7634	196865964DB3D417B6BD4D586BCB7634	Pass
253548FFCA461C67C8CBC78CD59F4756	A0A1A2A3A5A6A7A8AAABACADAFB0B1B2B4B5B6B7B9BABBBCBEBFC0C1C3C4C5C6	60436AD45AC7D30D99195F815D98D2AE	60436AD45AC7D30D99195F815D98D2AE	Pass
FD7AD8D73B9B0F8CC41600640F503D65	C8C9CACBCDCECFD0D2D3D4D5D7D8D9DADCDDDEDFE1E2E3E4E6E7E8E9EBECEDEE	BB07A23F0B61014B197620C185E2CD75	BB07A23F0B61014B197620C185E2CD75	Pass
06199DE52C6CBF8AF954CD65830BCD56	F0F1F2F3F5F6F7F8FAFBFCFDFE01000204050607090A0B0C0E0F101113141516	5BC0B2850129C854423AFF0751FE343B	5BC0B2850129C854423AFF0751FE343B	Pass
F17C4FFE48E44C61BD891E257E725794	18191A1B1D1E1F20222324252728292A2C2D2E2F31323334363738393B3C3D3E	7541A78F96738E6417D2A24BD2BECA40	7541A78F96738E6417D2A24BD2BECA40	Pass
9A5B4A402A3E8A59BE6BF5CD8154F029	40414243454647484A4B4C4D4F50515254555657595A5B5C5E5F606163646566	B0A303054412882E464591F1546C5B9E	B0A303054412882E464591F1546C5B9E	Pass
79BD40B91A7E07DC939D441782AE6B17	68696A6B6D6E6F70727374757778797A7C7D7E7F81828384868788898B8C8D8E	778C06D8A355EEEE214FCEA14B4E0EEF	778C06D8A355EEEE214FCEA14B4E0EEF	Pass
D8CEAAF8976E5FBE1012D8C84F323799	90919293959697989A9B9C9D9FA0A1A2A4A5A6A7A9AAABACAEAFB0B1B3B4B5B6	09614206D15CBACE63227D06DB6BEEBB	09614206D15CBACE63227D06DB6BEEBB	Pass
3316E2751E2E388B083DA23DD6AC3FBE	B8B9BABBBDBEBFC0C2C3C4C5C7C8C9CACCCDCECFD1D2D3D4D6D7D8D9DBDCDDDE	41B97FB20E427A9FDBBB358D9262255D	41B97FB20E427A9FDBBB358D9262255D	Pass
8B7CFBE37DE7DCA793521819242C5816	E0E1E2E3E5E6E7E8EAEBECEDEFF0F1F2F4F5F6F7F9FAFBFCFEFE010103040506	C1940F703D845F957652C2D64ABD7ADF	C1940F703D845F957652C2D64ABD7ADF	Pass
F23F033C0EEBF8EC55752662FD58CE68	08090A0B0D0E0F10121314151718191A1C1D1E1F21222324262728292B2C2D2E	D2D44FCDAE5332343366DB297EFCF21B	D2D44FCDAE5332343366DB297EFCF21B	Pass
59EB34F6C8BDBACC5FC6AD73A59A1301	30313233353637383A3B3C3D3F40414244454647494A4B4C4E4F505153545556	EA8196B79DBE167B6AA9896E287EED2B	EA8196B79DBE167B6AA9896E287EED2B	Pass
DCDE8B6BD5CF7CC22D9505E3CE81261A	58595A5B5D5E5F60626364656768696A6C6D6E6F71727374767778797B7C7D7E	D6B0B0C4BA6C7DBE5ED467A1E3F06C2D	D6B0B0C4BA6C7DBE5ED467A1E3F06C2D	Pass
E33CF7E524FED781E7042FF9F4B35DC7	80818283858687888A8B8C8D8F90919294959697999A9B9C9E9FA0A1A3A4A5A6	EC51EB295250C22C2FB01816FB72BCAE	EC51EB295250C22C2FB01816FB72BCAE	Pass
27963C8FACDF73062867D164DF6D064C	A8A9AAABADAEAFB0B2B3B4B5B7B8B9BABCBDBEBFC1C2C3C4C6C7C8C9CBCCCDCE	ADED6630A07CE9C7408A155D3BD0D36F	ADED6630A07CE9C7408A155D3BD0D36F	Pass
77B1CE386B551B995F2F2A1DA994EEF8	D0D1D2D3D5D6D7D8DADBDCDDDFE0E1E2E4E5E6E7E9EAEBECEEEFF0F1F3F4F5F6	697C9245B9937F32F5D1C82319F0363A	697C9245B9937F32F5D1C82319F0363A	Pass
F083388B013679EFCF0BB9B15D52AE5C	F8F9FAFBFDFEFE00020304050708090A0C0D0E0F11121314161718191B1C1D1E	AAD5AD50C6262AAEC30541A1B7B5B19C	AAD5AD50C6262AAEC30541A1B7B5B19C	Pass
C5009E0DAB55DB0ABDB636F2600290C8	20212223252627282A2B2C2D2F30313234353637393A3B3C3E3F404143444546	7D34B893855341EC625BD6875AC18C0D	7D34B893855341EC625BD6875AC18C0D	Pass

D Program Code

The following pages contain the source code used to compile the two implementations.

QAES.c	-	High Speed Implementation (QuickAES)
AES.c	-	Low Memory Implementation

```
/*    FAST IMPLEMENTATION OF AES ENCRYPTION     ##
##    WRITTEN BY CHRIS FELDWICK – APRIL 2005    */

#include <math.h>

typedef unsigned char byte;

//Global Variables

byte NoRounds[3] = {10,12,14};
int Nk, Nr;

//Lookup Tables

byte LogTable[256], ALogTable[256], SBox[256], Rcon[15];
```

```
byte GFPolyMult( byte a, byte b)
{
  byte temp;
  int r=0;

  //Multiplication using shifting algorithm:

  while (a != 0)
  {
    if ((a & 1) != 0)
      r = r ^ b;
    temp = b & 0x80;
    b = b << 1;
    if (temp != 0)
      b = b ^ 0x1b;
    a = a >> 1;
  }

  //return the result of the multiplication

  return r;
}
```

```
getPlainText (byte array[4][4], char *text)
{
  long value;
  byte temp[1];
  byte * end;
  int i,j;

  //Read in string in 2's, convert to hex, put in state array

  for (i=0; i<4; i++){
    for (j=0; j<4; j++){
      temp[0] = text[(8*i)+(2*j)];
      temp[1] = text[(8*i)+(2*j)+1];
```

```
      //convert into a usable value
      array[j][i] =(int) strtol(temp, &end, 16);
    }
  }

}
```

```
getKeyText (byte array[4][8], char *text)
{
  long value;
  byte temp[1];
  byte * end;
  int i,j;

  //read in strings in 2's, convert to hex, put in key array

  for (i=0; i<Nk; i++){
    for (j=0; j<4; j++){
      temp[0] = text[(8*i)+(2*j)];
      temp[1] = text[(8*i)+(2*j)+1];

      //convert strings into usable values
      array[j][i] =(int) strtol(temp, &end, 16);
    }
  }

}
```

```
printState( byte array[4][4])
{
  //prints the current state onto screen

  int i,j;
  for (i=0; i<4; i++)
  {
    for (j=0; j<4; j++)
    {
      //Print values in hexadecimal characters:
      printf("%02X", array[j][i]);
    }

  }printf("\n");
}
```

```
copyArray( byte from[4][4], byte to[4][4])
{
  //Copies the contents of one array to another

  int i,j;
  for (i=0; i<4; i++){
    for (j=0; j<4; j++){
      to[i][j] = from[i][j];
    }
  }
}
```

```
byte LogMult(byte a, byte b)
{
  //Uses log tables to calculate the multiplication of two bytes
  //exception case - either values 0 then return 0

  if (a==0 || b==0){
    return 0;
  }

  //return the result of the multiplication

  return ALogTable[(LogTable[a]+LogTable[b])%255];

}
```

```
shiftRows( byte state[4][4])
{
  //Performs the shiftRow transform on the state array

  byte temp[4][4];
  int i,j,newcol;
  for (i=0; i<4; i++)
  {
    for (j=0; j<4; j++)
    {
      //calculate the shift needed
      newcol = (j+(4-i))%4;
      temp[i][newcol] = state[i][j];
    }
  }
  //copy contents of temp back into state.
  copyArray(temp, state);
}
```

```
mixColumns (byte state[4][4])
{
  //Performs MixColumns transform on State

  byte ab, cd, abcd;
  int j;
  for(j=0; j<4; j++){

    //Calculate convenient XORs for use below

    ab = state[0][j] ^ state[1][j];
    cd = state[2][j] ^ state[3][j];
    abcd = ab ^ cd;

    //Mix Cols Refined Calculations:
```

```
    state[3][j] = state[3][j] ^ abcd ^
LogMult(2,(state[3][j]^state[0][j]));
    state[1][j] = state[1][j] ^ abcd ^
LogMult(2,(state[1][j]^state[2][j]));
    state[0][j] = state[0][j] ^ abcd ^ LogMult(2,ab);
    state[2][j] = state[2][j] ^ abcd ^ LogMult(2,cd);

  }
}
```

```
byte extEuclid (byte a)
{
  //Finds the multiplicative inverse of the input byte
  //This is the Binary Extended Euclidean Algorithm

  int m;
  byte b,d;

  // Exception Case: Inverse of 0 defined as 0

  if (a==0){
    return 0;
  }

  m = 0x11b;
  b = 0; d = 1;

  //Keep looping until GCD is found
  while (m!=0){

    //Keep doing while multiple of 2
    while ((m%2)==0){

      m = m >> 1;

      if ((b%2)==0){
      b = b >> 1;
      }else{
      b = ((b^0x11b)>>1);
      }

    }

    //Keep doing while multiple of 2
    while ((a%2)==0){

      a = a >> 1;

      if ((d%2)==0){
      d = d >> 1;
      }
      else{
      d = ((d^0x11b)>>1);
      }

    }
    if (m>=a){
      m = m ^ a;
      b = b ^ d;
    }
    else{
      a = a ^ m;
      d = d ^ b;
    }
  }
  //Return the inverse of the original byte
  return d;
}
```

```
byte affTrans(byte a)
{
  //uses function byteBit to perform the affine transform on a byte

  int i;
  byte result, temp;
  result = 0;

  //calulate each bit of new byte individually

  for (i=0; i<8; i++){
    temp = byteBit(a,i) ^ byteBit(a,(i+4)%8) ^ byteBit(a,(i+5)%8) ^
byteBit(a,(i+6)%8) ^ byteBit(a,(i+7)%8);

    //add together the values of the new bits

    result = result + (byte)(pow(2,i)*temp);
  }

  //carry out final XOR operation and return new byte

  return (result ^ 0x63);
}
```

```
subByte(byte state[4][4])
{
  //uses the s-box to replace each element of the state array

  int i,j;
  for (i=0; i<4; i++){
    for (j=0; j<4; j++){
      state[i][j] = SBox[state[i][j]];
    }
  }
}
```

```
int byteBit(int number, int bit)
{
  //Calculate if the relevant bit of the byte is 1 or 0
```

```
  return (number%(int)(pow(2,(bit+1))))>=pow(2,bit);
}
```

```
keyExpansion(byte key[4][8], byte W[][4])
{ //creates the expanded key from the input key
  int i,j,a,b,c,d;
  byte temp[4];
  byte tem;

  //set up values which act as pointers to top item in each word
  a=0; b=0;

  //Copy across first Nk words

  for (j=0; j<Nk; j++){

    for(i=0; i<4; i++){
      W[a+i][b] = key[i][j];
    }

    //move pointers to next word
    b++;
    if (b==4){
      b=0; a=a+4;
    }

  }

  //Recursively define the remaining key expansion

  for (i=Nk; i<((Nr+1)*4); i++){

    //copy previous word into temp using pointers a,b
    if (b==0){
      d=3;
      c = a-4;
    }else{
      d = b-1;
      c = a;
    }

    for (j=0; j<4; j++){
      temp[j] = W[c+j][d];
    }



    //apply additional changes if multiple of Nk

    if (i%Nk == 0){

      //SubBytes Routine
      for (j=0; j<4; j++){
      temp[j] = SBox[temp[j]];
      }

      //Rotating Word
      tem = temp[0];
      for (j=0; j<3; j++){
      temp[j]=temp[j+1];
      }
      temp[3] = tem;

      //XOR With Rcon

      temp[0] = temp[0] ^ Rcon[(i%Nk)-1];

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

      //Slightly different procedure when Nk > 6

      for (j=0; j<4; j++){
      temp[j] = SBox[temp[j]];
      }

    }

    //Move temp back into KeyExpansion Whilst XOR i-4th Word

    //use c and d as pointers to top item in the i-4th word
    c=a-4; d=b;


    for(j=0; j<(Nk-4); j++){
      if(d==0){
      d=3;
      c=c-4;
      }else{
      d--;
      }
    }

    for (j=0; j<4; j++){
      W[a+j][b] = temp[j] ^ W[c+j][d];
    }

    //move pointers a,b to next word and continue...
    b++;
    if (b==4){
      b=0; a=a+4;
    }
  }
}
```

```
addRoundKey(byte state[4][4], int round, byte ExpKey[44][4])
{
  //XOR state with sub-array of the ExpandedKey

  int i,j;

  for (i=0; i<4; i++){
    for (j=0; j<4; j++){
      state[i][j] = state[i][j] ^ ExpKey[(round)*4 + i][j];
    }
```

```
  }
}
```

```
initialise()
{
  int i,temp;

  //Place powers of x into Rcon
  temp = 0x01;
  Rcon[0] = temp;


  for (i=1; i<15; i++){
    temp=GFPolyMult(temp, 2);
    Rcon[i] = temp;
  }

  //Set up SubByte Lookup Table

  for(i=0; i<256; i++){
    SBox[i]=affTrans(extEuclid(i));
  }

  //Calculate Log Tables using 3 as generator

  temp = 1;
  for(i=0; i<256; i++){
    LogTable[temp] = i;
    ALogTable[i] = temp;
    temp  = GFPolyMult(temp,3);
  }


}
```

```
main(int argc, char *argv[])
{
  //create variables

  int i,j,k,temp;
  byte W[60][4];
  byte Key[4][8];
  byte state[4][4];

  //try to catch wrongly input data:

  if (argc != 4){
    printf("syntax:\tQAES [key size in bits] [plaintext (32 Hex
Chars)] [key]\n");
    exit(0);
  }

  sscanf(argv[1],"%d", &temp);

  if (temp!=128 && temp!=192 && temp!= 256){
    printf("Invalid Key Size.  Must be 128, 192 or 256. Type QAES for
syntax help\n");
    exit(1);
  }

  if (strlen(argv[2])!=32 || strlen(argv[3])!= (temp/4)){
    printf("Invalid length of plaintext or key, Type QAES for syntax
help\n");
    exit(1);
  }

  //Initialise LookUp Tables;
  initialise();

  //calulate global variables
  sscanf(argv[1],"%d", &Nk);
  Nk = Nk/32;
  Nr = NoRounds[(Nk/2)-2];

  //Retrieve Key and Perform Expansion
  getKeyText(Key, argv[3]);
  keyExpansion(Key, W);

  //Retrieve Plaintext
  getPlainText(state, argv[2]);

  //AddRoundKey before first round
  addRoundKey(state,0,W);

  //Standard Rounds
  for (i=1; i<Nr; i++){

    subByte(state);
    shiftRows(state);
    mixColumns(state);
    addRoundKey(state,i,W);

  }

  //Final Round:
  subByte(state);
  shiftRows(state);
  addRoundKey(state,Nr,W);

  //print out result to screen
  printState(state);


}
```

```
/*    LOW MEMORY IMPLEMENTATION OF AES ENCRYPTION
      WRITTEN BY CHRIS FELDWICK - APRIL 2005    */

#include <string.h>
#include <math.h>

typedef unsigned char byte;

//Global Variables

byte NoRounds[3] = {10,12,14};
int Nk, Nr;
```

```
byte GFPolyMult( byte a, byte b)
{
  byte temp;
  int r=0;

  //Multiplication using shifting algorithm:

  while (a != 0)
  {
    if ((a & 1) != 0)
      r = r ^ b;
    temp = b & 0x80;
    b = b << 1;
    if (temp != 0)
      b = b ^ 0x1b;
    a = a >> 1;
  }

  //return the result of the multiplication

  return r;
}
```

```
getPlainText (byte array[4][4], char *text)
{
  long value;
  byte temp[1];
  byte * end;
  int i,j;

  //Read in string in 2's, convert to hex, put in state array

  for (i=0; i<4; i++){
    for (j=0; j<4; j++){
      temp[0] = text[(8*i)+(2*j)];
      temp[1] = text[(8*i)+(2*j)+1];

      //convert into a usable value
      array[j][i] =(int) strtol(temp, &end, 16);
    }
  }

}
```

```
getKeyText (byte array[4][8], char *text)
{
  long value;
  byte temp[1];
  byte * end;
  int i,j;

  //read in strings in 2's, convert to hex, put in key array

  for (i=0; i<Nk; i++){
    for (j=0; j<4; j++){
      temp[0] = text[(8*i)+(2*j)];
      temp[1] = text[(8*i)+(2*j)+1];

      //convert strings into usable values
      array[j][i] =(int) strtol(temp, &end, 16);
    }
  }

}
```

```
copyRow( byte from[4], byte to[4][4], int row)
{
  int j;

    for (j=0; j<4; j++){
      to[row][j] = from[j];
    }

}
```

```
printState( byte array[4][4])
{
  //prints the current state onto screen

  int i,j;
  for (i=0; i<4; i++)
  {
    for (j=0; j<4; j++)
    {
      //Print values in hexadecimal characters:
      printf("%02X", array[j][i]);
    }

  }printf("\n");
}
```

```
shiftRows( byte state[4][4])
{
  //Performs the shiftRow transform on the state array
  byte temp[4];
  int i,j,newcol;
  for (i=0; i<4; i++)
  {
    for (j=0; j<4; j++)
    {
      //calculate the shift needed
      newcol = (j+(4-i))%4;
      temp[newcol] = state[i][j];
    }

    //copy contents of temp row back into state.
    copyRow(temp, state, i);
  }

}
```

```
mixColumns (byte state[4][4])
{
  //Performs MixColumns transform on State

  byte ab, cd, abcd;
  int j;
  for(j=0; j<4; j++){

    //Calculate convenient XORs for use below

    ab = state[0][j] ^ state[1][j];
    cd = state[2][j] ^ state[3][j];
    abcd = ab ^ cd;

    //Mix Cols Refined Calculations:

    state[3][j] = state[3][j] ^ abcd ^
GFPolyMult(2,(state[3][j]^state[0][j]));
    state[1][j] = state[1][j] ^ abcd ^
GFPolyMult(2,(state[1][j]^state[2][j]));
    state[0][j] = state[0][j] ^ abcd ^ GFPolyMult(2,ab);
    state[2][j] = state[2][j] ^ abcd ^ GFPolyMult(2,cd);

  }
}
```

```
byte extEuclid (byte a)
{
  //Finds the multiplicative inverse of the input byte
  //This is the Binary Extended Euclidean Algorithm

  int m;
  byte b,d;

  // Exception Case: Inverse of 0 defined as 0

  if (a==0){
    return 0;
  }
```

```
  m = 0x11b;
  b = 0; d = 1;

  //Keep looping until GCD is found
  while (m!=0){

    //Keep doing while multiple of 2
    while ((m%2)==0){

      m = m >> 1;

      if ((b%2)==0){
      b = b >> 1;
      }else{
      b = ((b^0x11b)>>1);
      }

    }

    //Keep doing while multiple of 2
    while ((a%2)==0){

      a = a >> 1;

      if ((d%2)==0){
      d = d >> 1;
      }
      else{
      d = ((d^0x11b)>>1);
      }

    }

    if (m>=a){
      m = m ^ a;
      b = b ^ d;
    }
    else{
      a = a ^ m;
      d = d ^ b;
    }

  }
  //Return the inverse of the original byte
  return d;
}
```

```
byte affTrans(byte a)
{
  //uses function byteBit to perform the affine transform on a byte

  int i;
  byte result, temp;
  result = 0;

  //calulate each bit of new byte individually
  for (i=0; i<8; i++){
    temp = byteBit(a,i) ^ byteBit(a,(i+4)%8) ^ byteBit(a,(i+5)%8) ^
byteBit(a,(i+6)%8) ^ byteBit(a,(i+7)%8);

    //add together the values of the new bits

    result = result + (byte)(pow(2,i)*temp);
  }

  //carry out final XOR operation and return new byte

  return (result ^ 0x63);
}
```

```
subByte(byte state[4][4])
{
  //performs SubByte transform on the fly (Without S-box)

  int i,j;
  for (i=0; i<4; i++){
    for (j=0; j<4; j++){
      state[i][j] = affTrans(extEuclid(state[i][j]));
    }
  }
}
```

```
int byteBit(int number, int bit)
{
  //Calculate if the relevant bit of the byte is 1 or 0

  return (number%(int)(pow(2,(bit+1))))>=pow(2,bit);
}
```

```
keyExpansion(byte key[4][8], byte W[][4])
{
  //creates the expanded key from the input key

  int i,j,a,b,c,d;
  byte temp[4];
  byte tem;

  //set up values which act as pointers to top item in each word
  a=0; b=0;

  //Copy across first Nk words

  for (j=0; j<Nk; j++){

    for(i=0; i<4; i++){
      W[a+i][b] = key[i][j];
    }

    //move pointers to next word
```

```
    b++;
    if (b==4){
      b=0; a=a+4;
    }

  }

  //Recursively define the remaining key expansion

  for (i=Nk; i<((Nr+1)*4); i++){

    //copy previous word into temp
    if (b==0){
      d=3;
      c = a-4;
    }else{
      d = b-1;
      c = a;
    }

    for (j=0; j<4; j++){
      temp[j] = W[c+j][d];
      //printf("[%02x]\n",temp[j]);
    }



    //apply additional changes if multiple of Nk

    if (i%Nk == 0){

      //SubBytes Routine
      for (j=0; j<4; j++){
      temp[j] = affTrans(extEuclid(temp[j]));
      }

      //Rotating Word
      tem = temp[0];
      for (j=0; j<3; j++){
      temp[j]=temp[j+1];
      }
      temp[3] = tem;

      //XOR With Rcon
      tem = 0x01;
      for(j=1; j<(i/Nk); j++){
      tem = GFPolyMult(tem, 2);
      }

      temp[0] = temp[0] ^ tem;

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

      //Slightly different when Nk > 6

      for (j=0; j<4; j++){
      temp[j] = affTrans(extEuclid(temp[j]));
      }
    }

    //Move temp back into KeyExpansion Whilst XOR i-4th Word

    //use c and d as pointers to top item in the i-4th word
    c=a-4; d=b;
    for(j=0; j<(Nk-4); j++){
      if(d==0){
      d=3;
      c=c-4;
      }else{
      d--;
      }
    }

    for (j=0; j<4; j++){
      W[a+j][b] = temp[j] ^ W[c+j][d];
    }

    //move pointers a,b to next word and continue...
    b++;
    if (b==4){
      b=0; a=a+4;
    }
  }
}
```

```
addRoundKey(byte state[4][4], int round, byte ExpKey[44][4])
{
  //XOR state with sub-array of the ExpandedKey

  int i,j;

  for (i=0; i<4; i++){
    for (j=0; j<4; j++){
      state[i][j] = state[i][j] ^ ExpKey[(round)*4 + i][j];
    }
  }
}
```

```
main(int argc, char *argv[])
{
  //create variables
  int i,j,k,temp;
  byte W[60][4];
  byte Key[4][8];
  byte state[4][4];

  //Try to catch wrongly input data

  if (argc != 4){
    printf("syntax:\tAES [key size in bits] [plaintext (32 Hex
Chars)] [key]\n");
    exit(0);
  }
```

```
  sscanf(argv[1],"%d", &temp);

  if (temp!=128 && temp!=192 && temp!= 256){
    printf("Invalid Key Size.  Must be 128, 192 or 256. Type AES for
syntax help\n");
    exit(1);
  }

  if (strlen(argv[2])!=32 || strlen(argv[3])!= (temp/4)){
    printf("Invalid length of plaintext or key, Type AES for syntax
help\n");
    exit(1);
  }

  //calulate global variables
  sscanf(argv[1],"%d", &Nk);
  Nk = Nk/32;
  Nr = NoRounds[(Nk/2)-2];

  //Retrieve Key and Perform Expansion
  getKeyText(Key, argv[3]);
  keyExpansion(Key, W);

  //Retrieve Plaintext
  getPlainText(state, argv[2]);

  //AddRoundKey before first round
  addRoundKey(state,0,W);

  //Standard Rounds
  for (i=1; i<Nr; i++){

    subByte(state);
    shiftRows(state);
    mixColumns(state);
    addRoundKey(state,i,W);

  }

  //Final Round
  subByte(state);
  shiftRows(state);
  addRoundKey(state,Nr,W);

  //print out result to screen
  printState(state);
}
```