# On "Secured" Encryption Schemes

A dissertation submitted in partial fulfillment
of the requirements of M.Tech.(Computer Science)
degree of Indian Statistical Institute, Calcutta
by

**Ganapathi CH**

under the supervision of

**Prof. Bimal Kumar Roy**
**Indian Statistical Institute**
**Kolkata-700 035.**

**20 July 2001**

**Indian Statistical Institute**

**203, Barrackpore Trunk Road,**

**Kolkata-700 035.**

**Certificate of Approval**

This is to certify that this thesis titled **On "Secured" Encryption Schemes** submitted by **Ganapathi CH** towards partial fulfillment of requirements for the degree of M.Tech in Computer Science at Indian Statistical Institute, Kolkata embodies the work done under my supervision.

[illegible]

Bimal Kumar Roy, 20.7.01
Professor,
Applied Statistics Unit,
Indian Statistical Institute,
Kolkata-700 035.

# Acknowledgements

I take pleasure in thanking Prof. Bimal Roy for his friendly guidance throughout the dissertation period. His pleasant and encouraging words have always kept my sagging spirits up.

I would also like to express my sincere gratitude to Mr. Shantanu Bhattacharya of Cognizant Technology Solutions for agreeing to share a few words with me which helped me in giving a right shape to my work.

I would like to thank Ms. Mahua Bagchi and others at Cognizant Technology Solutions Academy for providing the required facilities during my work.

## Contents

## 1 Introduction

Data encryption (cryptography) is utilized in various applications and environments. In general, cryptography is used to protect data while it is being communicated between two points or while it is stored in a medium vulnerable to physical theft. Communication security provides protection to data by enciphering it at the transmitting point and deciphering it at the receiving point. File security provides protection to data by enciphering it when it is recorded on a storage medium and deciphering it when it is read back from the storage medium. In the first case, the key must be available at the transmitter and receiver simultaneously during communication. In the second case, the key must be maintained and accessible for the duration of the storage period.

The most well-known symmetric encryption algorithm is the Data Encryption Standard (**DES**). It defines a block cipher with 64-bit blocks and 56-bit keys. Because of its small key size exhaustion of the .DES (i.e., breaking a **DES** encrypted ciphertext by trying all possible keys) has become increasingly more feasible with technology advances. Following a recent hardware based **DES** key exhaustion attack, National Institute of Standards and Technology (**NIST**) [13] no onger supports the use of single **DES** for many applications and suggested the use of **triple DES** (i.e. applying single **DES** 3 times). **Triple DES** is currently the most widely used encryption standard.

In 1997, **NIST** initiated a process to select a symmetric-key encryption algorithm, to be called Advanced Encryption Standard (**AES**), which is atleast as secure as **triple DES** and capable of protecting sensitive government information well into the twenty-first century. **NIST** expects that the algorithm will be used by the U.S. Government and, on a voluntary basis, by the private sector.

The organization of this report is as follows: In section 2 we review **DES**, **triple DES** and their cryptanalysis. In section 3 we sketch the development process for the **AES** and a brief overview of the five finalists of **AES**. In section 4 we discuss the proposed **AES** (**Rijndael**) and a software implementation by Dr.Brian Gladman [7]. In the section 5 we briefly discuss the main features of **Encryption Modes with Almost Free Message Integrity** for **AES** suggested by Charanjit S. Jutla [11] and our implementation of that mode for **AES**. The mode has two variations 1) Integrity Aware Cipher Block Chaining Mode (**IACBC**) and 2) Integrity Aware Parallelizable Mode (**IAPM**). The Appendix contains the C implementation of **IAPM**. Minor changes are required in this code to implement **IACBC**.

## 2 DES, triple DES and their cryptanalysis

### 2.1 Data Encryption Standard (DES)

The algorithm is designed to encipher and decipher blocks of data consisting of 64 bits under control of a 64-bit key[1] of which 56 bits are randomly generated and used directly by the algorithm. The other 8 bits, which are not used by the algorithm, may be used for error detection. The 8 error detecting bits are set to make the parity of each 8-bit byte of the key odd, i.e., there is an odd number of "1"s in each 8-bit byte. Deciphering must be accomplished by using the same key as for enciphering, but with the schedule of addressing the key bits altered so that the deciphering process is the reverse of the enciphering process.

#### *Enciphering*

A block to be enciphered is subjected to an initial permutation $IP$. The permuted input block is then the input to a complex key-dependent computation. The output of that computation, called the preoutput, is then subjected to a permutation which is the inverse of the initial permutation $IP^{-1}$.

The computation which uses the permuted input block as its input to produce the preoutput block consists, but for a final interchange of blocks, of 16 iterations of a calculation that is described below in terms of the cipher function $f$ which operates on two blocks, one of 32 bits and one of 48 bits, and produces a block of 32 bits.

Let the 64 bits of the input block to an iteration consist of a 32 bit block $L$ followed by a 32 bit block $R$. We then denote this by $LR$. Let $K$ be a block of 48 bits chosen from the 64-bit key. Then the output $L^{'}R^{'}$ of an iteration with input $LR$ is defined by:

$$\begin{aligned} L^{'} &= R \\ R^{'} &= L \oplus f(R,K) \end{aligned} \tag{1}$$

where $\oplus$ denotes bit-by-bit addition modulo 2.

As remarked before, the input of the first iteration of the calculation is the permuted input block. If $L^{'}R^{'}$ is the output of the 16th iteration then $R^{'}L^{'}$ is the preoutput block. At each iteration a different block $K$ of key bits is chosen from the 64-bit key designated by $KEY$.

With more notation we can describe the iterations of the computation in more detail. Let $KS$ be a function which takes an integer n in the range from 1 to 16 and a 64-bit block $KEY$ as input and yields as output a 48-bit block $K_n$ which is a permuted selection of bits from $KEY$. That is

$$K_n = KS(n, KEY) \tag{2}$$

with $K_n$ determined by the bits in 48 distinct bit positions of $KEY$. $KS$ is called the key schedule because the block $K$ used in the $n^{'}$th iteration of (1) is the block $K_n$ determined by (2).

As before, let the permuted input block be $LR$. Finally, let $L_0$ and $R_0$ be respectively $L$ and $R$ and for $1 \leq n \leq 16$ let

$$\begin{aligned} L_n &= R_{n-1} \\ R_n &= L_{n-1} \oplus f(R_{n-1}, K_n) \end{aligned}$$

The preoutput block is then $R_{16}L_{16}$. The key schedule produces the 16 $K_n$ for the algorithm.

#### *Deciphering*

From (1) it follows that:

$$\begin{aligned} R &= L^{'} \\ L &= R^{'} \oplus f(L^{'},K) \end{aligned}$$

Consequently, to ***decipher*** it is only necessary to apply the ***very same algorithm to an enciphered message block,*** taking care that at each iteration of the computation ***the same block of key bits K is used*** during decipherment as was used during the encipherment of the block.

[1]Blocks are composed of bits numbered from left to right, i.e., the left most bit of a block is bit one.

### 2.1.1 Differential Cryptanalysis of DES

Differential Cryptanalysis is a potent cryptanalytic technique introduced by Biham and Shamir [2]. Differential cryptanalysis is designed for the study and attack of DES-like cryptosystems. A DES-like cryptosystem is an iterated cryptosystem which relies on conventional cryptographic techniques such as substitution and diffusion.

Differential cryptanalysis is a chosen-plaintext/chosen-ciphertext cryptanalytic attack. Cryptanalysts choose a large number of ciphertexts $Y$, $Y'$ whose corresponding plaintexts $X$, $X'$ satisfy a known difference $D = X \oplus X'$, where $\oplus$ is componentwise XOR. They then study the difference between the members of the corresponding pair of ciphertexts. Statistics of the plaintext pair-ciphertext pair differences can yield information about the key used in encryption. In the basic Biham-Shamir attack, $2^{47}$ such plaintext pairs are required to determine the key for DES. Substantially fewer pairs are required if DES is truncated to 6 or 8 rounds. In these cases, the actual key can be recovered in a matter of minutes using a few thousand pairs. For full DES this attack is impractical because it requires so many known plaintexts.

The general form of the new variant of the attack can be summarized in the following way: Given a characteristic with probability p and signal to noise ratio $S/N$ for a cryptosystem with $k$ key bits, we can apply a memoryless attack which encrypts $\frac{2}{p}$ chosen plaintexts in the data collection phase and has complexity of $\frac{2^k}{S/N}$ trial encryptions during the data analysis phase. The number of chosen plaintexts can be reduced to $\frac{1}{p}$ by using appropriate metastructures, and the effective time complexity can be reduced by a factor of $f \leq 1$ if a tested key can be discarded by carrying out only a fraction f of the rounds. Therefore, memoryless attacks can be mounted whenever $p > 2^{1-k}$ and $S/N > 1$.

The performance of the new attack for various numbers of rounds is summarized in the following table:

| Rounds | Chosen Plaintexts | Analysed Plaintexts | complexity of Analysis |
|---|---|---|---|
| 8 | $2^{14}$ | 4 | $2^{9}$ |
| 9 | $2^{24}$ | 2 | $2^{32}$ |
| 10 | $2^{34}$ | $2^{14}$ | $2^{15}$ |
| 11 | $2^{31}$ | 2 | $2^{32}$ |
| 12 | $2^{31}$ | $2^{21}$ | $2^{21}$ |
| 13 | $2^{39}$ | 2 | $2^{32}$ |
| 14 | $2^{39}$ | $2^{29}$ | $2^{29}$ |
| 15 | $2^{47}$ | $2^{7}$ | $2^{37}$ |
| 16 | $2^{47}$ | $2^{36}$ | $2^{37}$ |

This specific attack is not directly applicable to plaintexts consisting of ASCII characters since such plaintexts cannot give rise to the desired XOR differences. By using several other iterative characteristics one can attack the full 16-round DES with a pool of about $2^{49}$ chosen plaintexts (out of the $2^{56}$ possible ASCII plaintexts).

### 2.1.2 Efficient DES Key Search

Despite recent improvements in analytic techniques for attacking the Data Encryption Standard (DES), exhaustive key search remains the most practical and efficient attack. Key search is becoming alarmingly practical. In [16] *Michael J. Wiener* describes how to build an exhaustive DES key search machine for $1 million that can find a key in 3.5 hours on average. The design for such a machine is described in detail in [16] for the purpose of assessing the resistance of DES to an exhaustive attack. The design is based on mature technology to avoid making guesses about future capabilities. With this approach, DES keys can be found one to two orders of magnitude faster than other recently proposed designs. The basic machine design can be adapted to attack the standard DES modes of operation for a small penalty in running time. The issues of development cost and machine reliability are examined as well. In light of this work, it would be prudent in many applications to use DES in a triple-encryption mode.

## 2.2 Triple Data Encryption Algorithm (TDEA)

Let $E_K(I)$ and $D_K(I)$ represent the DES encryption and decryption of $I$ using DES key $K$ respectively. Each TDEA encryption/decryption operation (as specified in **ANSI X9.52**) is a compound operation of DES encryption and decryption operations. The following operations are used:

1. TDEA encryption operation: the transformation of a 64-bit block $I$ into a 64-bit block $O$ that is defined as follows:
$$O = E_{K3}(D_{K2}(E_{K1}(I))).$$
2. TDEA decryption operation: the transformation of a 64-bit block $I$ into a 64-bit block $O$ that is defined as follows:
$$O = D_{K1}(E_{K2}(D_{K3}(I))).$$

The standard specifies the following keying options for bundle $(K_1,\ K_2,\ K_3)$

1. Keying Option 1: $K_1,\ K_2$ and $K_3$ are independent keys;
2. Keying Option 2: $K_1$ and $K_2$ are independent keys and $K_3 = K_1$;
3. Keying Option 3: $K_1 = K_2 = K_3$.

Note that Keying Option 3 ($K_1 = K_2 = K_3$) is nothing but single DES.

### 2.2.1 Attacking Triple Encryption

The standard technique to attack triple encryption is the meet-in-the-middle (MITM) attack. In [10] Stephen Lucks presents more efficient attacks. He presents 3 differnet attacks whose performance is given below when compared to MITM. In the table below $l$ is the no.of pairs of plaintext/ciphertext required for the attack.

| attack | l | memory | steps | single encrypions |
|---|---|---|---|---|
| MITM | 3 | $2^{56}$ | $2^{111}$ | $2^{111}$ |
| operation | | | | |
| optimized | $2^{45}$ | $2^{56}$ | $2^{109.2}$ | $2^{109.2}$ |
| (variant) | $(2^{45})$ | $2^{56}$ | $2^{108.2}$ | $2^{108.2}$ |
| encryption | 1 | $l * 2^k$ | $2^{2k}$ | $3 * 2^{3k-s} + 2^{k+s}/l$ |
| optimized | $2^{16}$ | $2^{72}$ | $2^{112}$ | $2^{106}$ |
| | $2^{24}$ | $2^{80}$ | $2^{112}$ | $3 * 2^{104}$ |
| | $2^{32}$ | $2^{88}$ | $2^{112}$ | $3 * 2^{104}$ |
| advanced | 1 | $l * 2^k$ | $2^{2k+1}$ | $l * 2^{2k+1} + 2^{k+s+1}/l$ |
| | $2^{16}$ | $2^{72}$ | $2^{113}$ | $2^{105}$ |
| | $2^{24}$ | $2^{80}$ | $2^{113}$ | $3 * 2^{97}$ |
| | $2^{32}$ | $2^{88}$ | $2^{113}$ | $3 * 2^{90}$ |

Note that in practice a single encryption is an exceptionally complex step when compared to common operations like comparisons and table look-ups.

Based on today's technology, neither MITM nor any of Lucks's attacks constitutes a practical way to break triple DES.

## 3 Development of the Advanced Encryption Standard

In 1997, the National Institute of Standards and Technology (NIST) initiated a process to select a symmetric-key encryption algorithm capable of protecting sensitive (unclassified) government information well into the twenty-first century. NIST expects that the algorithm will be used by the U.S. Government and, on a voluntary basis, by the private sector. In 1998, NIST announced the acceptance of fifteen candidate algorithms and requested the assistance of the cryptographic research community in analyzing the candidates. This analysis included an initial examination of the security and efficiency characteristics for each algorithm. NIST reviewed the results of this preliminary research and selected MARS, RC6$^{TM}$, Rijndael, Serpent and Twofish as finalists. Having reviewed further public analysis of the finalists, NIST selected Rijndael as the proposed AES algorithm.

### 3.1 Overview of the Finalists

The five finalists are iterated block ciphers: they specify a transformation that is iterated a number of times on the data block to be encrypted or decrypted. Each iteration is called a round, and the transformation is called the round function. Each finalist also specifies a method for generating a series of keys from the original user key; the method is called the *key schedule*, and the generated keys are called subkeys. The round functions take distinct subkeys as input along with the data block.

For each finalist, the very first and last cryptographic operations are some form of mixing of subkeys with the data block. Such mixing of secret subkeys prevents an adversary who does not know the keys from even beginning to encrypt the plaintext or decrypt the ciphertext. Whenever this subkey mixing does not naturally occur as the initial step of the first round or the final step of the last round, the finalists specify the subkey mixing as an extra step called pre- or post-whitening. Below is a summary of each of the finalist candidates in alphabetical order:

**MARS** [3] has several layers: key addition as pre-whitening, 8 rounds of unkeyed forward mixing, eight rounds of keyed forward transformation, 8 rounds of keyed backwards transformation, eight rounds of unkeyed backwards mixing, and key subtraction as post-whitening. The 16 keyed transformations are called the cryptographic core. The unkeyed rounds use two 8x32 bit S-boxes, addition, and the XOR operation. In addition to those elements, the keyed rounds use 32-bit key multiplication, data-dependent rotations, and key addition. Both the mixing and the core rounds are modified Feistel rounds in which one fourth of the data block is used to alter the other three fourths of the data block. MARS was submitted by the International Business Machines Corporation (IBM).

**RC6** [14] is a parameterized family of encryption ciphers that essentially use the Feistel structure; 20 rounds were specified for the AES submission. The round function of RC6 uses variable rotations that are regulated by a quadratic function of the data. Each round also includes 32-bit modular multiplication, addition, XOR, and key addition. Key addition is also used for pre- and post-whitening. RC6 was submitted by RSA Laboratories.

**Rijndael** [5] is a substitution-linear transformation network[1] with 10, 12 or 14 rounds, depending on the key size. A data block to be processed using Rijndael is partitioned into an array of bytes, and each of the cipher operations is byte-oriented. Rijndaels round function consists of four layers. In the first layer, an 8x8 S-box is applied to each byte. The second and third layers are linear mixing layers, in which the rows of the array are shifted, and the columns are mixed. In the fourth layer, subkey bytes are XORed into each byte of the array. In the last round, the column mixing is omitted. Rijndael was submitted by Joan Daemen (Proton World International) and Vincent Rijmen (Katholieke Universiteit Leuven).

**Serpent** [1] is a substitution-linear transformation network consisting of 32 rounds. Serpent also specifies non-cryptographic initial and final permutations that facilitate an alternative mode of implementation called the bitslice mode. The round function consists of three layers: the key XOR operation, 32 parallel applications of one of the eight specified 4x4 S-boxes, and a linear transformation.

[1]Ciphers that process the entire data block in parallel during each round using substitution and linear transformations

In the last round, a second layer of key XOR replaces the linear transformation. Serpent was submitted by Ross Anderson (University of Cambridge), Eli Biham (Technion), and Lars Knudsen (University of California San Diego).

**Twofish** [15] is a Feistel network with 16 rounds. The Feistel structure is slightly modified using 1-bit rotations. The round function acts on 32-bit words with four key-dependent 8x8 S-boxes, followed by a fixed 4x4 maximum distance separable matrix over GF($2^8$), a pseudo-Hadamard transform, and key addition. Twofish was submitted by Bruce Schneier, John Kelsey, and Niels Ferguson (Counterpane Internet Security, Inc.), Doug Whiting (Hi/fn, Inc.), David Wagner (University of California Berkeley), and Chris Hall (Princeton University).

## 3.2 Evaluation Criteria

The evaluation criteria were divided into three major categories: 1) Security, 2) Cost, and 3) Algorithm and Implementation Characteristics. Security was the most important factor in the evaluation and encompassed features such as resistance of the algorithm to cryptanalysis, soundness of its mathematical basis, randomness of the algorithm output, and relative security as compared to other candidates.

Cost was a second important area of evaluation that encompassed licensing requirements, computational efficiency (speed) on various platforms, and memory requirements.

The third area of evaluation was algorithm and implementation characteristics such as flexibility, hardware and software suitability, and algorithm simplicity. Flexibility includes the ability of an algorithm:

- To handle key and block sizes beyond the minimum that must be supported,
- To be implemented securely and efficiently in many different types of environments, and
- To be implemented as a stream cipher, hashing algorithm, and to provide additional cryptographic services.

It must be feasible to implement an algorithm in both hardware and software, and efficient firmware implementations were considered advantageous. The relative simplicity of an algorithms design was also an evaluation factor.

Comparisons of the finalists in some of the criterias which lead NIST to select **Rijndael** as proposed AES algorithm are as follows:

### 3.2.1 General Security

Security was the foremost concern in evaluating the finalists. NIST relied on the public security analysis conducted by the cryptographic community. **No attacks have been reported against any of the finalists, and no other properties have been reported that would disqualify any of them.**

The only attacks that have been reported to date are against simplified variants of the algorithms: the number of rounds is reduced or simplified in other ways. Attacks on reduced rounds do not necessarily imply anything about the security of the original algorithms. Almost all of these attacks require more than $2^{30}$ encryptions of chosen plaintexts; in other words, more than a billion encryptions, and in some cases far more are required. Even if a single key were used this many times, it might be impractical for an adversary to collect so much information.

### 3.2.2 Other Security Observations

Many observations have been offered about various properties that might impact the security of the finalists. Because the implications of these observations are generally subjective, they did not play a significant role in NIST's selection.

### 3.2.3 Summary of Security Characteristics of the Finalists

As no general attacks against any of the finalists is known the determination of the level of security provided by the finalists is largely guesswork, as in the case of any unbroken cryptosystem. **MARS**, **Serpent and Twofish** appears to have a high security margin **Rijndael** and **RC6** appears to have an adequate security margin.

### 3.2.4 Summary of Speed on General Software Platforms

The table below gives an overall performance of the finalists on the various platforms when using 128-keys.

| | Enc/Dec | Key Setup |
|---|---|---|
| MARS | II | II |
| RC6 | I | II |
| Rijndael | I | I |
| Serpent | III | II |
| Twofish | II | III |

### 3.2.5 A Case Study

All finalists were implemented on a high-end smart card which was equipped with a Z80 microprocessor (8 bits), a coprocessor, 48 Kbytes of ROM, and 1 Kbyte of RAM. The Z80 can execute logical instructions, 1-bit shifts or rotations, addition, and subtraction. The coprocessor is useful in handling modular multiplications, completing a multiplication within the execution time of a Z80 instruction. The coprocessor can also be called upon for other arithmetic or logical operations, if advantageous.

The results are summarized in the following table. This table clearly indicates that Rijndael is superior in every respect, within the scope of the present study.

| | RAM | ROM | ENC | KEY | TIME |
|---|---|---|---|---|---|
| MARS | 572 | 5468 | 45 | 21 | 67 |
| RC6 | 156 | 1060 | 34 | 138 | 173 |
| RIJN | 66 | 980 | 25 | 10 | 35 |
| SERP | 164 | 3937 | 71 | 147 | 219 |
| TWOF | 90 | 2808 | 31 | 28 | 60 |

Legend:

| | | |
|---|---|---|
| RAM | = | Total RAM in bytes. |
| ROM | = | Total ROM in bytes. |
| ENC | = | Time for encryption of one 128-bit block, in units of 1000 cycles. |
| KEY | = | Time for key scheduling, in units of 1000 cycles. |
| TIME | = | Encryption + key scheduling, in units of 1000 cycles. |

### 3.2.6 Comaparison of all Hardware Results

Among all finalists **Rijndael** allows high throughput designs for a variety of architectures, namely basic, pipelined and unrolled implementations. When fully unrolled, 128-bit Rijndael has the lowest single block encryption latency of any of the finalists and, therefore, the highest throughput for feedback mode encryption. In standard architecture implementations, the throughput of 128-bit implementations is also at or near the top. Most of the studies did not consider other key sizes. However, since Rijndael adds additional rounds for larger key sizes, throughput in the standard architecture or unrolled implementations falls with larger key sizes, but still remains good. For fully pipelined implementations, area requirements increase with larger key sizes, but throughput is unaffected. Rijndael has good performance in fully pipelined implementations, giving it non-feedback throughput performance that is second only to Serpent. Efficiency is generally very good.

## 4 Advanced Encryption Standard (Rijndael)

The three criteria taken into account in the design of Rijndael are the following:

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

### 4.1 Specification

Rijndael is an iterated symmetric block cipher with a variable block length and a variable key length. The block length and the key length can be independently specified to **128, 192** or **256 bits**. But for the standard the block length is **128 bits**.

For both its Cipher and Inverse Cipher, the AES algorithm uses a round function that is composed of four different byte-oriented transformations:

1) byte substitution using a substitution table (S-box)
2) shifting rows of the State array by different offsets
3) mixing the data within each column of the State array, and
4) adding a Round Key to the State.

#### 4.1.1 The State, the Cipher Key and the number of rounds

The different transformations operate on the intermediate result, called the State. The State can be pictured as a rectangular array of bytes. This array has four rows, the number of columns is denoted by ***Nb*** and is equal to the block length divided by 32.

The Cipher Key is similarly pictured as a rectangular array with four rows. The number of columns of the Cipher Key is denoted by ***Nk*** and is equal to the key length divided by 32.

If the input block consists of the following bytes:

$$a_0a_1a_2\dots a_{15}$$

then the block is mapped into the State as below

| | | | |
|---|---|---|---|
| $a_0$ | $a_4$ | $a_8$ | $a_{12}$ |
| $a_1$ | $a_5$ | $a_9$ | $a_{13}$ |
| $a_2$ | $a_6$ | $a_{10}$ | $a_{14}$ |
| $a_3$ | $a_7$ | $a_{11}$ | $a_{15}$ |

At the end the output block is extracted from the State in the same order.

The number of rounds is denoted by ***Nr*** and depends on the values ***Nb*** and ***Nk*** as shown below:

| $Nr$ | $Nk = 4$ | $Nk = 6$ | $Nk = 8$ |
|---|---|---|---|
| $Nb = 4$ | 10 | 12 | 14 |
| $Nb = 6$ | 12 | 12 | 14 |
| $Nb = 8$ | 14 | 14 | 14 |

### 4.2 Cipher

The cipher AES algorithm consists of

- an initial Round Key addition;
- ***Nr***-1 Rounds;
- a final round.

The Cipher is described in the following pseudo code:

```
Cipher(byte in[4 * Nb], byte out[4 * Nb], word w[Nb * (Nr + 1)])
begin
   byte state[4,Nb]

   state = in

   AddRoundKey(state, w)

   for round = 1 to Nr - 1 step 1
      SubBytes(state)
      ShiftRows(state)
      MixColumns(state)
      AddRoundKey(state, w + round * Nb)
   end for

   SubBytes(state)
   ShiftRows(state)
   AddRoundKey(state, w + Nr * Nb)

   out = state
end
```

The individual transformations -`SubBytes()`, `ShiftRows()`, `MixColumns()`, and `AddRoundKey()` process the State and are described in the following subsections. The array **w**[] contains the key schedule, which is described in Sec. 4.3. As shown above, all ***Nr*** rounds are identical with the exception of the final round, which does not include the `MixColumns()` transformation.

### 4.2.1 SubBytes() Transformation

The SubBytes() Transformation is a non-linear byte substitution that operates independently on each byte of the State using a substitution table (S-box). This S-box is invertible and is constructed by composing two transformations:

1. Take the multiplicative inverse in the finite field GF($2^8$); the element {00} which has no inverse is mapped to itself.

2. Apply an affine (over GF(2)) transformation defined by:

$$b_i' = b_i \oplus b_{(i+4)mod8} \oplus b_{(i+5)mod8} \oplus b_{(i+6)mod8} \oplus b_{(i+7)mod8} \oplus c_i \tag{3}$$

   for $0 \le i < 8$, where $b_i$ is the $i^{th}$ bit of the byte, and $c_i$ is the $i^{th}$ bit of a byte c with the value {63} or {01100011}.

### 4.2.2 ShiftRows() Transformation

In the `ShiftRows()` transformation, the bytes in the last three rows of the State are cyclically shifted over different numbers of bytes (offsets). The first row, Row 0, is not shifted.

Specifically, the `ShiftRows()` transformation proceeds as follows:

$$s'_{r,c} = s_{r,(c+shift(r,Nb))modNb} \quad for\ 0 < r < 4 \quad and \quad 0 \le c < \boldsymbol{Nb},$$

where the shift value *shift(r,Nb)* depends on the row number, r, as follows(recall that ***Nb*** = 4):

$$shift(1,4) = 1; \quad shift(2,4) = 2; \quad shift(3,4) = 3. \tag{4}$$

### 4.2.3 MixColumns() Transformation

The MixColumns() transformation operates on the State column-by-column, treating each column as a four-term polynomial[1]. The columns are considered as polynomials over GF($2^8$) and multiplied modulo $x^4 + 1$ with a fixed polynomial a(x), given by

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

### 4.2.4 AddRoundKey() Transformation

In the AddRoundKey() transformation, a Round Key is added to the State by a simple bitwise XOR operation. Each Round Key consists of ***Nb*** words from the key schedule (described in Sec. 4.3). Those ***Nb*** words are each added into the columns of the State, such that

$$[s'_{0,c}, s'_{1,c}, s'_{2,c}, s'_{3,c}] = [s_{0,c}, s_{1,c}, s_{2,c}, s_{3,c}] \oplus [w_{round*Nb+c}] \qquad for\ 0 \le c < \boldsymbol{Nb},$$

where $[w_i]$ are the key schedule words described in Sec. 4.3, and *round* is a value in the range $0 \le round \le Nr$. In the Cipher, the initial Round Key addition occurs when $round = 0$, prior to the first application of the round function. The application of the AddRoundKey() transformation to the ***Nr*** rounds of the Cipher occurs when $1 \le round \le \boldsymbol{Nr}$.

## 4.3 Key Expansion

The AES algorithm takes the Cipher Key, ***K***, and performs a Key Expansion routine to generate a key schedule. The Key Expansion generates a total of ***Nb*** (***Nr*** + 1) words: the algorithm requires an initial set of ***Nb*** words, and each of the ***Nr*** rounds requires ***Nb*** words of key data. The resulting key schedule consists of a linear array of 4-byte words, denoted $[w_i]$, with i in the range $0 \le i < \boldsymbol{Nb}(\boldsymbol{Nr}+1)$.

The expansion of the input key into the key schedule proceeds is as follows: the first ***Nk*** words of the expanded key are filled with the Cipher Key. Every following word, **w[i]**, is equal to the XOR of the previous word, **w[i-1]**, and the word ***Nk*** positions earlier, **w[i-Nk]**. For words in positions that are a multiple of ***Nk***, a transformation is applied to **w[i-1]** prior to the XOR, followed by an XOR with a round constant, Rcon[i]. This transformation consists of a cyclic shift of the bytes in a word (RotWord()), followed by the application of a table lookup to all four bytes of the word (SubWord()).

The Key Expansion routine for 256-bit Cipher Keys (***Nk*** = 8) is slightly different than for 128- and 192-bit Cipher Keys. If ***Nk*** = 8 and i-4 is a multiple of ***Nk***, then SubWord() is applied to **w[i-1]** prior to the XOR.

SubWord() is a function that takes a four-byte input word and applies the S-box to each of the four bytes to produce an output word. The function RotWord() takes a word $[a_0, a_1, a_2, a_3]$ as input, performs a cyclic permutation, and returns the word $[a_1, a_2, a_3, a_0]$. The round constant word array, Rcon[i], contains the values given by $[x^{i-1}, \{00\}, \{00\}, \{00\}]$, with $x^{i-1}$ being powers of x (x is denoted as {02}) in the field GF($2^8$), and note that here i starts at 1, not 0.

## 4.4 Inverse Cipher

The Cipher transformations in Sec. 4.2 can be inverted and then implemented in reverse order to produce a straightforward Inverse Cipher for the AES algorithm. The individual transformations used in the Inverse Cipher - InvShiftRows(), InvSubBytes(), InvMixColumns(), and AddRoundKey() process the State and are described in the following subsections.

[1]Four-term polynomials can be defined - with coefficients that are finite field elements - as:

$$a(x) = a_3\,x^3 + a_2\,x^2 + a_1\,x + a_0$$

which will be denoted as a word in the form $[a_0,\ a_1,\ a_2,\ a_3]$.

### 4.4.1 InvShiftRows() Transformation

`InvShiftRows()` is the inverse of the `ShiftRows()` transformation. The bytes in the last three rows of the State are cyclically shifted over different numbers of bytes (offsets). The first row, Row 0, is not shifted. The bottom three rows are cyclically shifted by ***Nb-shift(r,Nb)*** bytes, where the shift value ***shift(r,Nb)*** depends on the row number, and is given in equation (4)(see Sec. 4.2.2).

### 4.4.2 InvSubBytes() Transformation

`InvSubBytes()` is the inverse of the byte substitution transformation, in which the inverse S-box is applied to each byte of the State. This is obtained by applying the inverse of the affine transformation (3)(see Sec. 4.2.1) followed by taking the multiplicative inverse in GF($2^8$).

### 4.4.3 InvMixColumns() Transformation

`InvMixColumns()` is the inverse of the `MixColumns()` transformation. `InvMixColumns()` operates on the State column-by-column, treating each column as a four-term polynomial as described in Sec. 4.2.3. The columns are considered as polynomials over GF($2^8$) and multiplied modulo $x^4 + 1$ with a fixed polynomial $a^{-1}(x)$, given by

$$a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\}$$

### 4.4.4 Inverse of the AddRoundKey() Transformation

`AddRoundKey()`, which was described in Sec. 4.2.4, is its own inverse, since it only involves an application of the XOR operation.

### 4.4.5 Equivalent Inverse Cipher

In the straightforward Inverse Cipher the sequence of the transformations differs from that of the Cipher, while the form of the key schedules for encryption and decryption remains the same. However, several properties of the AES algorithm allow for an Equivalent Inverse Cipher that has the same sequence of transformations as the Cipher (with the transformations replaced by their inverses). This is accomplished with a change in the key schedule.

```
    EqInvCipher(byte in[4 * Nb], byte out[4 * Nb], word dw[Nb * (Nr + 1)])
    begin
       byte state[4,Nb]

       state = in

       AddRoundKey(state, dw + Nr * Nb)

       for round = Nr - 1  to 1 step -1
          InvSubBytes(state)
          InvShiftRows(state)
          InvMixColumns(state)
          AddRoundKey(state, dw + round * Nb)
       end for

       InvSubBytes(state)
       InvShiftRows(state)
       AddRoundKey(state, dw)

       out = state
end
```

The word array dw[] in the above Pseudo code for the Equivalent Inverse Cipher contains the modified decryption key schedule. For the Equivalent Inverse Cipher, the following pseudo code is added at the end of the Key Expansion routine described in Sec. 4.3.

```
for i = 0 to (Nr + 1) * (Nb - 1) step 1
   dw[i] = w[i]
end for

for rnd = 1 to (Nr - 1) step 1
   InvMixColumns(dw + rnd * Nb)
end for
```

### 4.4.6 Attcks on Rijndael

The standard techniques of differential and linear cryptanalysis can be adapted to be used against ***Rijndael***. Because of the way matrix multiplication works, and because in GF($2^8$), all the coefficients of the Mix Column matrix (i.e all numbers from 1 to 255) have reciprocals, a specific attack, originally developed for use against its predecessor Square [4], called the "Square attack", can be used as well.

If one uses 256 blocks of chosen plaintext, where every byte but one is held constant, and that one is given all 256 possible values, then after one round of ***Rijndael***, four bytes will go through all 256 possible values, and the rest of the bytes will remain constant. After a second round, sixteen bytes will each go through all 256 possible values, without a single duplicate, in the encipherment of the 256 blocks of chosen plaintext. (For a 128-bit block, this is every byte; for larger blocks, the rest of the bytes will remain constant.) This interesting property, although not trivial to exploit, can be used to impose certain conditions on the key when one additional round, before or after the two rounds involved, is present.

The possibility of this attack was first noted by the developers of Square and Rijndael themselves, and is explained in the paper that initially described Square [4].

## 4.5 An Implementation of Rijndael

Here we shall briefly discuss the C, C++ implementation of AES algorithm by Dr.Brian Gladman[1]. For code see [7].

This implementation is designed to provide both fixed and dynamic block and key lengths and can also run with either big or little *endian* internal byte order. It inputs block and key lengths in bytes with the legal values being 16(128 bits), 24(192 bits) and 32(256 bits).
The implementation can be used using 3 functions parameterized as follows:

1. aes_ret set_key(const byte key[], const word key_length, const enum aes_key direction, aes *cx)
2. aes_ret encrypt(const byte input_blk[], byte output_blk[], const aes *cx)
3. aes_ret decrypt(const byte input_blk[], byte output_blk[], const aes *cx)

Here *key[]* is an array of bytes of size *KEY_LENGTH*, *input_blk[]* and *output_blk[]* are arrays of bytes of length *BLOCK_LENGTH*. *direction* can be *enc*, *dec* or *both* where *enc* and *dec* stand for encryption and decryption respectively. *aes* is structure which will hold information like direction, expanded key etc.

For *encryption/decryption/both* first the key expansion is done by calling *set_key()* with *direction enc/dec/both* and then encryption of a plaintext block can be done by calling *encrypt()* with *input_blk* containing the plaintext and *cx* the structure set in *set_key()* and the *output_blk* will contain the ciphertext on successful encryption; the decryption can be done with *input_blk* having ciphertext and *output_blk* containing plaintext on successful decryption.

[1]Dr. Brian Gladman undertakes consultancy work in the area of open systems security. He implemented all 5 AES finalists in C, C++ for NIST.

## 5 Proposed Modes of Operation

NIST expects to issue a guideline or recommendation in the Spring of 2001 for some encryption modes, updating the four DES modes and possibly including some form of Counter mode. Several proposals for modes were presented and discussed at a public workshop on October 20, 2000, in Baltimore, Maryland. The public has proposed more than ten modes of operation for consideration. To promote further analysis of all the modes submitted for consideration, NIST plans to hold a second public workshop on August 24, 2001, in Goleta, California. NIST proposes three general categories of criteria: security, performance, and mode/implementation characteristics. As with the evaluation of the AES candidate algorithms, security is the most important criterion.

Our implementation of *Encryption Modes with Almost Free Message Integrity* proposed by *Charanjit S. Jutla* of IBM T.J. Watson Research Center, in C using Dr.Brian Gladman's implementation of AES is given in Appendix. The proposal has two new modes of operation for symmetric key block cipher algorithms. The features of the these modes which lead to our selection of these for implementation are discussed in the following section. For complete specification of the modes see [11].

### 5.1 IACBC and IAPM

The main feature distinguishing the two proposed modes from existing modes is that along with providing confidentiality of the message, they also provide message integrity. In other words, the new modes are not just modes of operation for encryption, but modes of operation for authenticated encryption. The new modes achieve the additional property with little extra overhead.

One of the new modes **Integrity Aware Parallelizable Mode(IAPM)** is highly parallelizable. In fact, the parallelizable mode has critical paths of only two block cipher invocations. By one estimate, a hardware implementation of this mode on a single board (housing 1000 block cipher units) achieves terabits/sec ($10^{12}$ bits/sec) of authenticated encryption. Moreover, there is no penalty for doing a serial implementation of this mode.

The new modes also come with **proofs of security**, assuming that the underlying block ciphers are secure. For confidentiality, the modes achieve the same provable security bounds as **CBC**. For authentication, the modes achieve the same provable security bounds as **CBC-MAC**. For proofs of security see [6]

The non-parallelizable mode **Integrity Aware Cipher Block Chaining Mode (IACBC)** is similar to the CBC mode. It differs from the CBC mode in that the output is whitened (XORed) with a pairwise independent random sequence. The pairwise independent sequence can be generated with little overhead. It is this whitening with the pairwise independent sequence that assures message integrity.

The parallelizable mode removes the chaining from the above mode, and instead does an input whitening (in addition to the output whitening) with a pairwise independent sequence. Thus, it becomes similar to the **ECB** mode. However, with the input whitening with the pairwise independent sequence the new mode has provable security similar to **CBC** (Note: **ECB** does not have security guarantees like **CBC**).

Both the parallelizable mode and the non-parallelizable mode come in two flavors. These flavors refer to how the pairwise independent sequence is generated. In one mode, the pairwise independent sequence is generated by a subset construction. In another mode, the pairwise independent sequence is generated by (ai+b) modulo a fixed prime number. There will be one standard prime number for each bit-size block cipher. Thus, for 64 bit block ciphers the prime could be $2^{64} - 257$. For 128 bit block ciphers, the prime could be $2^{128} - 159$.

For intellectual property rights of these modes see [13]
We have chosen the subset construction flavor in our implementation.

## References

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

[2] Eli Biham and Adi Shamir, *Differential Cryptanalysis of the full 16-round DES*, Advances in Cryptography, Crypto '92.

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

[4] J. Daemen, L. R. Knudsen and V. Rijmen, *"The Block Cipher Square"*, Fast Software Encryption, FSE'97, LNCS 1267, Springer-Verlag, 1997.

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

[6] *http://eprint.iacr.org/2000/039.ps*

[7] *http://fp.gladman.plus.com/cryptography_technology/rijndael/*

[8] FIPS PUB 46-3, *DATA ENCRYPTION STANDARD (DES)*, Reaffirmed 1999 October 25.

[9] FIPS PUB 140-2, *ADVANCED ENCRYPTION STANDRD (AES)*, DRAFT, 2001.

[10] Stephen Lucks, *Attacking Triple Encryption*, Fast Software Encryption (1998), Springer LNCS.

[11] Charanjit S. Jutla, Encryption Modes with Almost Free Message Integrity, Proposed Modes of Operation for AES.

[12] James Nechvatal, *Report on the Development of the Advanced Encryption Standard (AES)*, October 2, 2000.

[13] *http://www.nist.gov/aes/*

[14] R. Rivest, et al., *The RC6$^{TM}$ Block Cipher*, AES algorithm submission, June 1998.

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

[16] Michael J. Wiener, *Efficient DES Key Search*, Technical Report TR-244, Carleton University, 1994.

# Appendix

## C code implementing IAPM

```
/************************************************************************/
/* Example of the use of the AES (Rijndael) algorithm for file        */
/* encryption.  Note that this is an example application, it is       */
/* not intended for real operational use.  The Command line is:       */
/*                                                                    */
/* aesiapm input_filename out_file_name [d/e] hex_key1 hex_key2       */
/*                                                                    */
/* where e gives encryption and d decryption of the input file        */
/* into the output file using the given hexadecimal key strings       */
/* Key strings are hexadecimal sequences of 32, 48 or 64 digits       */
/************************************************************************/
/*                  inlclude standard header files                    */
#include "aes.h"  /* include B.R.Gladman's implementation of AES */

#define New(p,n)     (p*)calloc(n,sizeof(p)) /* memory allocation */
#define Check(p)     if(!p) \       /* check for successful memory allocation */
                        {  printf("There is not enough memory!\n"); exit(-20);  }
#define Return(s,f,n)  {  printf(s,f); return n; }
#define OnError(s,f,n) {  printf(s,f); err = n; goto exit; }

void getBLOCK_SIZEbytesRandom(char r[])       /* sizeof(r) is BLOCK_LENGTH  */
{
    int i, j, random;                         /* fills r with random number */
    for(i = (BLOCK_SIZE -1)/2; i >= 0; i--) {
        random = rand();
        for(j = 1; j >= 0; j--)
            r[2*i+j] = bval(random,1-j);
    }
}
/* adds k to a BLOCK_SIZE number */
void add_k_to_BLOCK_SIZEbyteRandom(char b[], int k)
{
    int  i = (BLOCK_SIZE-1);

    if((unsigned char)(b[i]) > (0xff-k)) {
        b[i--] += k, k = 0;
        for(; i >= 0 && (unsigned char)(b[i]) > 0xfe; i--)
            b[i]++;
    }
    else {
        b[i] += k;
        return;
    }
    if(i >= 0)
        b[i]++;
}
/* generate pairwise independent sequence by subset construction */
char **pairwise_independent_sequence(char r[], int m, aes *aesK0)
{
  int     i,j,k;
  char    W[sizeof(int)*8][BLOCK_SIZE], tmp[BLOCK_SIZE], **S = New(char*, m+1); Check(S);

  for(i = 0; i < m+1; i++) {
      S[i] = New(char, BLOCK_SIZE); Check(S[i]);
  }

  encrypt(r,W[0],aesK0);

  strncpy(S[0], W[0], BLOCK_SIZE); strncpy(tmp, W[0], BLOCK_SIZE);

  k = (int)(ceil)(log((double)m+2)/log(2.0));

  for(i = 1; i <= k; i++) {
      add_k_to_BLOCK_SIZEbyteRandom(tmp, 1);
      encrypt(tmp, W[i++], aesK0);
  }
  for(i = 1; i <= m; i++) {
      j = i+1; k = 0;
      /* find the index of the least significant ON bit in (i+1) */
      while( !(j&1) ) {
          k++; j >>= 1;   /* increment k and right shift */
      }
      for(j = 0; j < BLOCK_SIZE; j++)
          S[i][j] = S[i-1][j] ^ W[k][j];
  }
  return S;
}

/* encrypt a file pointed by fin and output to fout. fn = filename */
int encfile(FILE *fin, FILE *fout, aes aesK[], char* fn)
{
   char          inbuf[BLOCK_SIZE], outbuf[BLOCK_SIZE], checksum[BLOCK_SIZE], **S;
   fpos_t        flen;
   int           i, j, l, m;

   srand( (unsigned)time( NULL ) );     /* initialise seed for rand()  */

   getBLOCK_SIZEbytesRandom(outbuf);    /*  set the IV for iapm mode   */

   fseek(fin, 0, SEEK_END);            /* get the length of the file   */
   fgetpos(fin, &flen);                /* and then reset to start      */
   fseek(fin, 0, SEEK_SET);

   if(fwrite(outbuf, 1, BLOCK_SIZE, fout) != BLOCK_SIZE)/* write the IV to the output   */
      Return("Error writing to output file: %s\n", fn, -8)

   l = BLOCK_SIZE == 16? 15: BLOCK_SIZE == 32? 31: 23;

   inbuf[0] = bval(rand(),0);           /* make a byte random and store the */
            /* length of the last block in the lower log(BLOCK_SIZE) bits */
   inbuf[0] = (char)(l==23?(flen%l|(inbuf[0]&~0x1f)):((char)flen&l)|(inbuf[0]&~l));
```

```
    m = (int)(ceil)((double)(flen+1)/BLOCK_SIZE) + 1; /* m-1 = no.of blocks  */
    /* generate m+1 pairwise independent random numbers */
    S = pairwise_independent_sequence(outbuf, m, &aesK[0]);

    memset(checksum, 0, BLOCK_SIZE); /* initialise checksum to zero     */

    for(i = 1; i < m; i++)            /* loop to encrypt the input file  */
    {               /* input 1st BLOCK_SIZE bytes to buf[1..BLOCK_SIZE] */
                    /*  on 1st round byte[0] is the length code         */
        if((j = fread(inbuf + BLOCK_SIZE - l, 1, l, fin)) < l)
        {   (l != BLOCK_SIZE)? j++:j; /* end of the input file reached? */
            while(j < BLOCK_SIZE)     /* adjust for 1st and later blocks*/
                inbuf[j++] = 0;       /* clear empty buffer positions if*/
        }                             /* any in the last block          */
        for(j = 0; j < BLOCK_SIZE; j++) {
            checksum[j] ^= inbuf[j];
            inbuf[j] ^= S[i][j];
        }
        encrypt(inbuf, outbuf, &aesK[1]);     /*   do the encryption    */
        for(j = 0; j < BLOCK_SIZE; j++)
            outbuf[j] ^= S[i][j];
        if(fwrite(outbuf, 1, BLOCK_SIZE, fout) != BLOCK_SIZE)
            Return("Error writing to output file: %s\n", fn, -9)
        l = BLOCK_SIZE;
    }
    for(j = 0; j < BLOCK_SIZE; j++)
        checksum[j] ^= S[m][j];

    encrypt(checksum, outbuf, &aesK[1]);        /*   encrypt checksum    */

    for(j = 0; j < BLOCK_SIZE; j++)
        outbuf[j] ^= S[0][j];
    if(fwrite(outbuf, 1, BLOCK_SIZE, fout) != BLOCK_SIZE)
        Return("Error writing to output file: %s\n", fn, -10)
    return 0;
}

int decfile(FILE *fin, FILE *fout, aes aesK[], char* ifn, char* ofn) {
    char    inbuf[BLOCK_SIZE], outbuf[BLOCK_SIZE], checksum[BLOCK_SIZE], **S;
    fpos_t flen;
    int     i, j, l, m, lastBlockLen;

    fseek(fin, 0, SEEK_END);         /* get the length of the file       */
    fgetpos(fin, &flen);             /* and then reset to start          */
    fseek(fin, 0, SEEK_SET);

    if(fread(inbuf, 1, BLOCK_SIZE, fin) != BLOCK_SIZE)     /* read IV   */
        Return("Error reading from input file: %s\n", ifn, -11)

    m = (int)((double)flen/BLOCK_SIZE)-1;/* m = no.of blocks in the decrypted file + IV */
    /* generate m+1 pairwise independent random numbers */
    S = pairwise_independent_sequence(inbuf, m, &aesK[0]);
    /* to recover the length of the last block */
    l = BLOCK_SIZE == 16? 15: BLOCK_SIZE == 32? 31: 23;

    memset(checksum, 0, BLOCK_SIZE); /* initialise checksum to zero     */

    for(i = 1; i < m; i++) {          /* loop to decrypt the input file  */
        if(fread(inbuf, 1, BLOCK_SIZE, fin) < BLOCK_SIZE)
            Return("\nThe input file %s is corrupt", ifn, -12)
        for(j = 0; j < BLOCK_SIZE; j++)
            inbuf[j] ^= S[i][j];
        decrypt(inbuf, outbuf, &aesK[1]);      /*   do the decryption     */
        for(j = 0; j < BLOCK_SIZE; j++) {
            outbuf[j] ^= S[i][j]; checksum[j] ^= outbuf[j];
        }
        if(i == 1)                        /* recover length of the last block */
        lastBlockLen = l==23? outbuf[0] & 0x1f: outbuf[0] & 1;
        if(i == m-1) {
            lastBlockLen += 1 - (l = (l == (BLOCK_SIZE - 1) ? 1 : 0));
            if((j = fwrite(outbuf + l, 1, lastBlockLen, fout)) != lastBlockLen)
                Return("Error writing to output file: %s\n", ofn, -13)
        } else
             if((j = fwrite(outbuf + BLOCK_SIZE - l, 1, l, fout)) != l)
                Return("Error writing to output file: %s\n", ofn, -14)
        l = BLOCK_SIZE;
    }
    if(fread(inbuf, 1, BLOCK_SIZE, fin) < BLOCK_SIZE)
        Return("\nThe input file %s is corrupt\n", ifn, -15)
    if(fgetc(fin) != EOF)
        Return("\nThe input file %s is corrupt\n", ifn, -16)
    for(j = 0; j < BLOCK_SIZE; j++)
        inbuf[j] ^= S[0][j];
    decrypt(inbuf, outbuf, &aesK[1]);          /*   decrypt last block    */
    for(j = 0; j < BLOCK_SIZE; j++) {
        outbuf[j] ^= S[m][j];
    }
    for(j = 0; j < BLOCK_SIZE; j++)           /*     Integrity check      */
        if(checksum[j] != outbuf[j])
            Return("Integrity check: failed!%s\n","",-17);
    printf("Integrity check: passed.\n");
    return 0;
}

int main(int argc, char *argv[])
{   FILE    *fin = 0, *fout = 0;
    char    *cp, ch, key[2][32];
    int     i, j, by, key_len, err = 0;
    aes     aesK[2];
    /*clock_t start, finish; double  duration; */

    if(argc != 6 || toupper(*argv[3]) != 'D' && toupper(*argv[3]) != 'E')
        OnError("usage: aesiapm input_filename out_file_name [d/e]
                        key1_in_hex key2_in_hex\n","",-1)
    for(i = 0; i < 2; i++) {
        cp = argv[i+4]; /* this is a pointer to the hexadecimal key digits  */
        j = 0;          /* this is a count for the input digits processed   */
        while(j < 65 && *cp)    /* the maximum key length is 32 bytes and   */
        {                       /* hence at most 64 hexadecimal digits      */
            ch = toupper(*cp++);            /* process a hexadecimal digit  */
            if(ch >= '0' && ch <= '9')
                by = (by << 4) + ch - '0';
            else if(ch >= 'A' && ch <= 'F')
                by = (by << 4) + ch - 'A' + 10;
            else                            /* error if not hexadecimal     */
                OnError("key must be in hexadecimal notation\n","",-2)
            /* store a key byte for each pair of hexadecimal digits         */
            if(j++ & 1)
                key[i][(j-1) / 2] = by & 0xff;
        }
        i == 0? key_len = j / 2:key_len;
        if( (i == 0) && (*cp) )
            OnError("The K0 key size is too long\n","",-3)
        else if( (i == 1) && ((key_len != j/2) || (*cp)) )
            OnError("The K1 key size must be same as K0 key size\n","",-4)
        else if(j < 32 || (j & 15))
            OnError("The length of key K%d must be 32,48 or 64 hexadecimal digits\n",i,-5)
    }
    if(!(fin = fopen(argv[1], "rb")))   /* try to open the input file */
        OnError("The input file: %s could not be opened\n", argv[1], -6)
    if(!(fout = fopen(argv[2], "wb")))  /* try to open the output file */
        OnError("The output file: %s could not be opened\n", argv[1], -7)
    /*start = clock();*/

    if(toupper(*argv[3]) == 'E')
    {                          /* encryption in Integrity Aware Parallelizable mode */
        set_key(key[0], key_len, enc, &aesK[0]);
        set_key(key[1], key_len, enc, &aesK[1]);

        err = encfile(fin, fout, aesK, argv[1]);
    }
    else {                     /* decryption in Integrity Aware Parallelizable mode */
        set_key(key[0], key_len, enc, &aesK[0]);
        set_key(key[1], key_len, dec, &aesK[1]);

        err = decfile(fin, fout, aesK, argv[1], argv[2]);
    }
exit:
    if(fout)
        fclose(fout);
    if(fin)
        fclose(fin);
    /* finish = clock(); duration = (double)(finish - start)/CLOCKS_PER_SEC;
    printf( "%2.6f seconds\n", duration ); */
    return err;
}
```