IMPLEMENTATION OF IMPROVED CBC MAC FOR ARBITRARY LENGTH MESSAGES

A DISSERTATION

Submitted in partial fulfillment of the requirements for the award of the degree

of

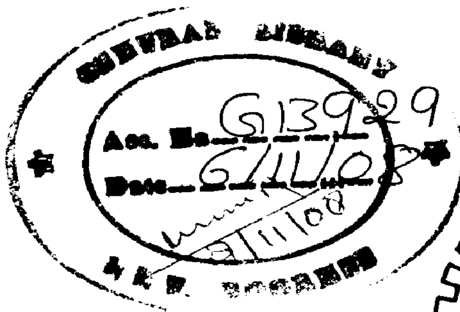
MASTER OF TECHNOLOGY

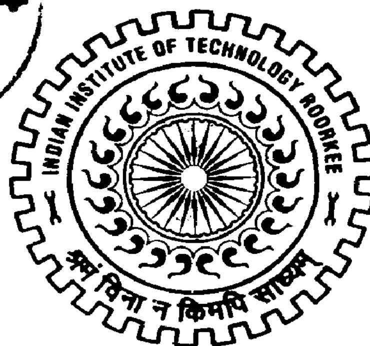
in

COMPUTER SCIENCE AND ENGINEERING

By

RANPISE SUDHIR RAMESH





DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY ROORKEE

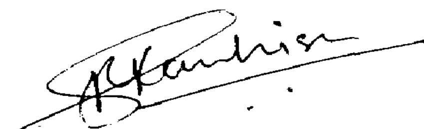
ROORKEE - 247 667 (INDIA)

JUNE, 2008

CANDIDATE'S DECLARATION

I hereby declare that the work, which is being presented in the dissertation entitled "*Implementation of improved CBC MAC for arbitrary length messages*" towards the partial fulfillment of the requirement for the award of the degree of **Master of Technology** in **Computer Science and Engineering** submitted in the Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, Roorkee (India) is an authentic record of my own work carried out during the period from June 2007 to June 2008, under the guidance of **Dr. Padam Kumar, Professor, Department of Electronics and Computer Engineering, IIT Roorkee.**

I have not submitted the matter embodied in this dissertation for the award of any other degree or diploma.



Date: 02/07/2008

Place: Roorkee

(RANPISE SUDHIR RAMESH)

CERTIFICATE

This is to certify that the above statement made by the candidate is correct to the best of my knowledge and belief.

Date: 02/07/2008

Place: Roorkee

(Dr. PADAM KUMAR)

Professor

Department of Electronics and Computer Engineering

IIT Roorkee – 247 667

ACKNOWLEDGEMENTS

I would like to take this opportunity to extend my heartfelt gratitude to my guide and mentor **Dr. Manoj Mishra**, Professor and **Dr. Padam Kumar**, Professor, Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, for their trust in my work, their esteemed guidance, regular source of encouragement and assistance throughout this dissertation work. I would state that the dissertation work would not have been in the present shape without their inspirational support and I consider myself fortunate to have done my dissertation under him.

I also extend my sincere thanks to **Dr. D. K. Mehra**, Professor, and Head of the Department of Electronics and Computer Engineering.

Finally, I would like to say that I am indebted to my parents for everything that they have done for me. All of this would have been impossible without their constant support.

RANPISE SUDHIR RAMESH

ABSTRACT

The CBC MAC (Cipher Block Chaining Message Authentication Code) is a well-known method to generate a message authentication code based on a block cipher. It is proved that the security of the CBC MAC for fixed message length mn bits, where n is the block length of the underlying block cipher E. However, it is well known that the CBC MAC is not secure unless the message length is fixed.

Therefore, several variants of CBC MAC have been proposed for variable length messages like EMAC, XCBC and TMAC.

Here, we propose and implement another variant for CBC MAC and prove its security for arbitrary length messages. The proposed mode takes only one key, K of a block cipher E. Previously, XCBC requires three keys, (k + 2n) bits in total, and TMAC requires two keys, (k + n) bits in total, where n denotes the block length of E. The saving of the key length makes the security proof of proposed mode substantially harder than those of XCBC.

CONTENTS

CANDIDATE'S DECLARATION.. i

ACKNOWLEDGEMENTS... ii

ABSTRACT.. iii

TABLE OF CONTENTS.. iv

CHAPTER 1: INTRODUCTION AND STATEMENT OF THE PROBLEM...... 1

1.1 Introduction 1

1.2 Statement of the Problem 2

1.3 Organization of the Dissertation 3

CHAPTER 2: BACKGROUND AND LITERATURE REVIEW........................... 4

2.1 Initialization Vector 4

2.2 Electronic Codebook (ECB) encryption mode 4

2.3 Cipher-block chaining (CBC) encryption mode 5

2.4 Cipher feedback (CFB) encryption mode 6

2.5 Output feedback (OFB) encryption mode 7

CHAPTER 3: AES ALGORITHM... 9

3.1 Information security and cryptography 9

3.2 Cryptographic goals 10

3.3 Inputs and Outputs 11

3.3.1 Bytes 11

3.3.2 Arrays of Bytes 13

3.3.3 The State 13

3.3.4 The State as an Array of Columns 14

3.4 Algorithm Specification 15

3.5 Cipher 16

3.5.1 SubBytes()Transformation 17

3.5.2 ShiftRows() Transformation 19
3.5.3 MixColumns() Transformation 20
3.5.4 AddRoundKey() Transformation 20
3.6 Key Expansion 21
3.7 Inverse Cipher 23
3.7.1 InvShiftRows() Transformation 24
3.7.2 InvSubBytes() Transformation 24
3.7.3 InvMixColumns() Transformation 24
3.7.4 Inverse of the AddRoundKey() Transformation 24
3.7.5 Equivalent Inverse Cipher 24

CHAPTER 4: IMPLEMENTATION.. 27
4.1 Pre-processing 27
4.2 Tag-generation 33
28
4.3 Tag-verification 30

CHAPTER 5: RESULTS AND DISCUSSION... 31
5.1 Results 31
5.2 Discussion 38

CHAPTER 6: CONCLUSION... 40

REFERENCES.. 41

CHAPTER 1
INTRODUCTION AND STATEMENT OF THE PROBLEM

1.1 Introduction

In cryptography, a block cipher operates on blocks of fixed length, often 64 or 128 bits where messages can be of any length. In block cipher, encrypting the same plaintext under the same key always produces the same output. To overcome the problem several modes of operation have been invented which allow block ciphers to provide confidentiality for messages of arbitrary length.

These modes are:

1) CCM mode (Counter with CBC-MAC)
2) EAX mode (Encryption with Associated Data)
3) GCM mode (Galois/Counter Mode)
4) OCB mode (Offset Codebook Mode)

CCM mode (Counter with CBC-MAC) is a mode of operation for cryptographic block ciphers. It is an authenticated encryption algorithm designed to provide both authentication and privacy. CCM mode only defines 128-bit block ciphers. CCM requires two cipher encryption operations for each block of encrypted and authenticated message and one encryption per each block of associated authenticated data. [7, 8]

EAX mode is a mode of operation for cryptographic block ciphers. It is an Authenticated Encryption with Associated Data (AEAD) algorithm designed to simultaneously protect both authentication and privacy of the message (Authenticated encryption) with a two-pass scheme, one pass for achieving privacy and one for authenticity for each block. Being a two-pass scheme, EAX mode is slower than a well-designed one-pass scheme based on the same primitives.

GCM mode (Galois/Counter Mode) is a mode of operation for symmetric key cryptographic block ciphers. It is an authenticated encryption algorithm designed to provide both authentication and privacy. GCM mode defines block ciphers with a block size of 128 bits. GCM requires one block cipher operation and one 128-bit multiplication in the Galois field per each block (128 bit) of encrypted and authenticated data. [7, 9]

OCB mode (Offset Codebook Mode) is a mode of operation for cryptographic block ciphers. OCB performance overhead is minimal comparing to classical, non-authenticating modes like CBC. OCB requires one block cipher encryption per each block of encrypted and authenticated message and one encryption per each block of additional associated data. There are also two extra encryptions required at the end of process. [9]

Block ciphers are often proposed with several variants, in terms of a different secret key size and corresponding number of rounds. The related-cipher attack model applicable to related ciphers in the sense that they are exactly identical to each other, differing only in the key size and most often also in the total number of rounds. Such related ciphers must have identical key schedules irrespective of their difference in the total number of rounds.

1.2 Statement of the Problem

There are several standard blocked cipher encryption modes for authentication but they all are having some advantages and disadvantages. There is a need to find out whether they all are secure or not. Also there is need to find out the opportunity to develop a better mode of operation for cryptographic block cipher.

The main objective of the thesis is to develop a better mode of operation for cryptographic block cipher.

1.3 Organization of the Dissertation

This report comprises of seven chapters including this chapter that introduces the topic and statement of the problem. The rest of the dissertation report is organized as follows:

Chapter 2 gives Background and literature review. It contains information about different types of cipher text modes. Initialization vector is also discussed in this chapter.

Chapter 3 gives details about the AES algorithm and various steps included in it. Algorithms for cipher and inverse cipher are discussed here. Chapter explains information security, cryptography and their goals.

Chapter 4 explains in details about the implementation of XMODE. Pre-processing, Tag generation and Tag verification are major the steps included in this chapter.

Chapter 5 shows the results with respect to AES 128, AES 192 and AES 256 and the key separation schemes used.

Chapter 6 concludes the work.

CHAPTER 2
BACKGROUND AND LITERATURE REVIEW

Block ciphers are often proposed with several variants, in terms of a different secret key size and corresponding number of rounds. The related-cipher attack model applicable to related ciphers in the sense that they are exactly identical to each other, differing only in the key size and most often also in the total number of rounds. Such related ciphers must have identical key schedules irrespective of their difference in the total number of rounds. Block cipher operates on blocks of fixed length, often 64 or 128 bits. Because messages may be of any length, and because encrypting the same plaintext under the same key always produces the same output, several modes of operation have been invented which allow block ciphers to provide confidentiality for messages of arbitrary length.

2.1 Initialization Vector (IV)

All these modes (except ECB) require an initialization vector, or IV -- a sort of 'dummy block' to kick off the process for the first real block, and also to provide some randomization for the process. There is no need for the IV to be secret, in most cases, but it is important that it is never reused with the same key. For CBC and CFB, reusing an IV leaks some information about the first block of plaintext, and about any common prefix shared by the two messages. For OFB and CTR, reusing an IV completely destroys security. In CBC mode, the IV must, in addition, be randomly generated at encryption time. [15]

2.2 Electronic Codebook (ECB) encryption mode

The earliest modes described in the literature (eg, ECB, CBC, OFB and CFB) provide only confidentiality, and do not ensure message integrity. Other modes have since been designed which ensure both confidentiality and message integrity, such as IAPM, CCM, EAX, GCM, and OCB modes. [12] Tweakable narrow-block

encryption (LRW) mode, and wide-block encryption (CMC and EME) modes, designed to securely encrypt sectors of a disk, are described in the article devoted to disk encryption theory.

The earliest modes described in the literature (eg, ECB, CBC, OFB and CFB) provide only confidentiality, and do not ensure message integrity. Other modes have since been designed which ensure both confidentiality and message integrity, such as CCM, EAX, GCM, and OCB modes. Tweakable narrow-block encryption (LRW) mode, and wide-block encryption (CMC and EME) modes, designed to securely encrypt sectors of a disk, are described in the article devoted to disk encryption theory.

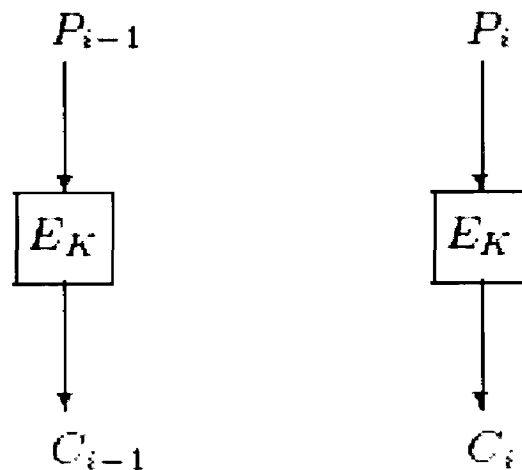


Fig. 2.1. ECB Mode Encryption

The ECB mode is the simplest, where each plaintext block, *Pi* is independently encrypted to a corresponding ciphertext block, *Ci* via the underlying block cipher, *Ek* keyed by secret key.

2.3 Cipher-block chaining (CBC) encryption mode

In the cipher-block chaining (CBC) mode, each block of plaintext is XORed with the previous ciphertext block before being encrypted. This way, each ciphertext block is dependent on all plaintext blocks processed up to that point. Also, to make each message unique, an initialization vector must be used in the first block.

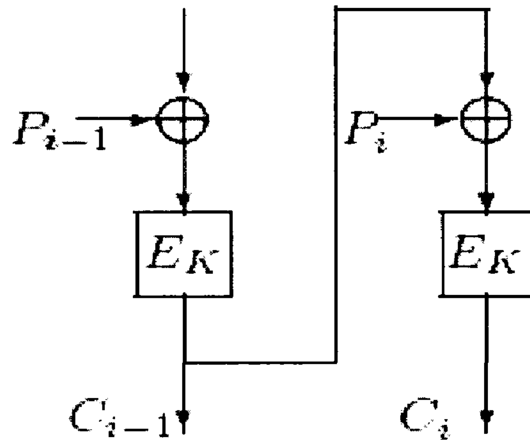


Figure 2.2: CBC mode encryption

CBC has been the most commonly used mode of operation. Its main drawbacks are that encryption is sequential (i.e., it cannot be parallelized), and that the message must be padded to a multiple of the cipher block size. One way to handle this last issue is through the method known as ciphertext stealing.

Note that a one-bit change in a plaintext affects all following ciphertext blocks, and a plaintext can be recovered from just two adjacent blocks of ciphertext. As a consequence, decryption *can* be parallelized, and a one-bit change to the ciphertext causes complete corruption of the corresponding block of plaintext, and inverts the corresponding bit in the following block of plaintext.

2.4 Cipher feedback (CFB) encryption mode

The cipher feedback (CFB) mode, a close relative of CBC, makes a block cipher into a self-synchronizing stream cipher. Operation is very similar; in particular, CFB decryption is almost identical to CBC encryption performed in reverse. [13,14]

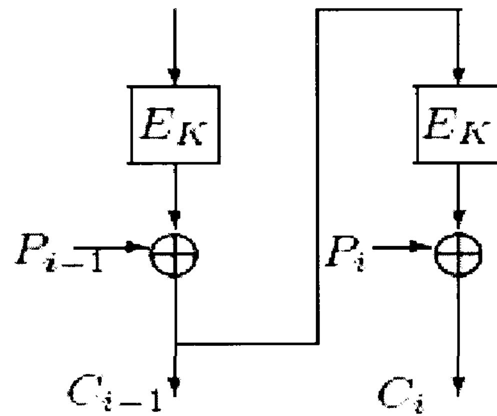


Figure 2.3: CFB mode encryption

Like CBC mode, changes in the plaintext propagate forever in the ciphertext, and encryption cannot be parallelized. Also like CBC, decryption can be parallelized. When decrypting, a one-bit change in the ciphertext affects two plaintext blocks: a one-bit change in the corresponding plaintext block, and complete corruption of the following plaintext block. Later plaintext blocks are decrypted normally.

Because each stage of the CFB mode depends on the encrypted value of the previous ciphertext XORed with the current plaintext value, a form of pipelining is possible, since the only encryption step which requires the plaintext is the final XOR. This is useful for applications that require low latency between the arrival of plaintext and the output of the corresponding ciphertext, such as certain applications of streaming media.

2.5 Output feedback (OFB) encryption mode

The output feedback (OFB) mode makes a block cipher into a synchronous stream cipher: it generates keystream blocks, which are then XORed with the plaintext blocks to get the ciphertext. Just as with other stream ciphers, flipping a bit in the ciphertext produces a flipped bit in the plaintext at the same location. This property allows many error correcting codes to function normally even when applied before encryption. [14, 15]

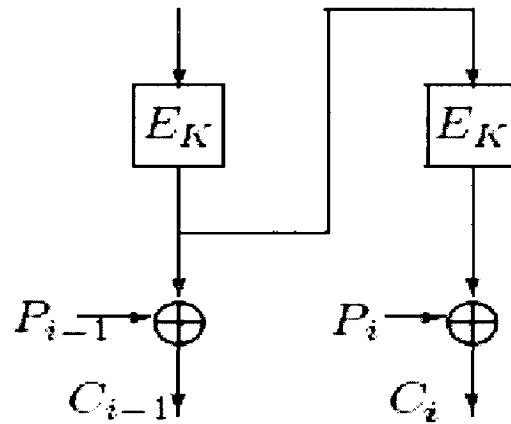


Figure 4: OFB mode encryption

Each output feedback block cipher operation depends on all previous ones, and so cannot be performed in parallel. However, because the plaintext or ciphertext is only used for the final XOR, the block cipher operations may be performed in advance, allowing the final step to be performed in parallel once the plaintext or ciphertext is available.

CHAPTER 3
AES ALGORITHM

AES algorithm is a symmetric block cipher that can encrypt (encipher) and decrypt (decipher) information. Encryption converts data to an unintelligible form called ciphertext; decrypting the ciphertext converts the data back into its original form, called plaintext. The AES algorithm is capable of using cryptographic keys of 128, 192, and 256 bits to encrypt and decrypt data in blocks of 128 bits.

The algorithm specified in this standard may be implemented in software, firmware, hardware, or any combination thereof. The specific implementation may depend on several factors such as the application, the environment, the technology used, etc. [1]

This standard specifies the Rijndael algorithm, a symmetric block cipher that can process data blocks of 128 bits, using cipher keys with lengths of 128, 192, and 256 bits. Rijndael was designed to handle additional block sizes and key lengths; however they are not adopted in this standard.

Throughout the remainder of this standard, the algorithm specified herein will be referred to as "the AES algorithm." The algorithm may be used with the three different key lengths indicated above, and therefore these different "flavors" may be referred to as "AES-128", "AES-192", and "AES-256".[2, 3]

3.1 Information security and cryptography

The concept of information will be taken to be an understood quantity. To introduce cryptography, an understanding of issues related to information security in general is necessary. Information security manifests itself in many ways according to the situation and requirement. Regardless of who is involved, to one degree or another,

all parties to a transaction must have confidence that certain objectives associated with information security have been met.

Conceptually, the way information is recorded has not changed dramatically over time. Whereas information was typically stored and transmitted on paper, much of it now resides on magnetic media and is transmitted via telecommunications systems, some wireless. What has changed dramatically is the ability to copy and alter information. One can make thousands of identical copies of a piece of information stored electronically and each is indistinguishable from the original. With information on paper, this is much more difficult. What is needed then for a society where information is mostly stored and transmitted in electronic form is a means to ensure information security, which is independent of the physical medium recording or conveying it and such that the objectives of information security rely solely on digital information itself. [4]

3.2 Cryptographic goals

Confidentiality is a service used to keep the content of information from all but those authorized to have it. Secrecy is a term synonymous with confidentiality and privacy. There are numerous approaches to providing confidentiality, ranging from physical protection to mathematical algorithms which render data unintelligible.

Data integrity is a service which addresses the unauthorized alteration of data. To assure data integrity, one must have the ability to detect data manipulation by unauthorized parties. Data manipulation includes such things as insertion, deletion, and substitution.

Authentication is a service related to identification. This function applies to both entities and information itself. Two parties entering into a communication should identify each other. Information delivered over a channel should be authenticated as to origin, date of origin, data content, time sent, etc. For these reasons this aspect of

cryptography is usually subdivided into two major classes: entity authentication and data origin authentication. Data origin authentication implicitly provides data integrity. [3]

Non-repudiation is a service which prevents an entity from denying previous commitments or actions. When disputes arise due to an entity denying that certain actions were taken, a means to resolve the situation is necessary. For example, one entity may authorize the purchase of property by another entity and later deny such authorization was granted. A procedure involving a trusted third party is needed to resolve the dispute.

3.3 Inputs and Outputs

The **input** and **output** for the AES algorithm each consist of **sequences of 128 bits** (digits with values of 0 or 1). These sequences will sometimes be referred to as **blocks** and the number of bits they contain will be referred to as their length. The **Cipher Key** for the AES algorithm is a **sequence of 128, 192 or 256 bits**. Other input, output and Cipher Key lengths are not permitted by this standard.

The bits within such sequences will be numbered starting at zero and ending at one less than the sequence length (block length or key length). The number i attached to a bit is known as its index and will be in one of the ranges $0 <= i < 128$, $0 <= i < 192$ or $0 <= i < 256$ depending on the block length and key length. [5]

3.3.1 Bytes

The basic unit for processing in the AES algorithm is a **byte,** a sequence of eight bits treated as a single entity. The input, output and Cipher Key bit sequences described in Sec. 4.1 are processed as arrays of bytes that are formed by dividing these sequences into groups of eight contiguous bits to form arrays of bytes (see Sec. 3.3.2). For an

input, output or Cipher Key denoted by a, the bytes in the resulting array will be referenced using one of the two forms, a_n or $a[n]$, where n will be in one of the following ranges:

Key length = 128 bits, $0 <= n < 16$; Block length = 128 bits, $0 <= n < 16$;
Key length = 192 bits, $0 <= n < 24$;
Key length = 256 bits, $0 <= n < 32$.

All byte values in the AES algorithm will be presented as the concatenation of its individual bit values (0 or 1) between braces in the order $\{b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0\}$. These bytes are interpreted as finite field elements using a polynomial representation:

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 = \sum_{i=0}^{7} b_i x^i .$$

For example, {01100011} identifies the specific finite field element $x^6 + x^5 + x + 1$.

It is also convenient to denote byte values using hexadecimal notation with each of two groups of four bits being denoted by a single character as in Fig. 1.

Bit Pattern	Character	Bit Pattern	Character	Bit Pattern	Character	Bit Pattern	Character
0000	0	0100	4	1000	8	1100	c
0001	1	0101	5	1001	9	1101	d
0010	2	0110	6	1010	a	1110	e
0011	3	0111	7	1011	b	1111	f

Figure 3.1. Hexadecimal representation of bit patterns.

Hence the element {01100011} can be represented as {63}, where the character denoting the four-bit group containing the higher numbered bits is again to the left.

3.3.2 Arrays of Bytes

Arrays of bytes will be represented in the following form:

$$a_0 a_1 a_2 ... a_{15}$$

The bytes and the bit ordering within bytes are derived from the 128-bit input sequence

$$input_0 \; input_1 \; input_2 \; ... \; input_{126} \; input_{127}$$

as follows:

$$a_0 = \{input_0, input_1, ..., input_7\};$$
$$a_1 = \{input_8, input_9, ..., input_{15}\};$$
$$\vdots$$
$$a_{15} = \{input_{120}, input_{121}, ..., input_{127}\}.$$

The pattern can be extended to longer sequences (i.e., for 192- and 256-bit keys), so that, in general,

$$a_n = \{input_{8n}, input_{8n-1}, ..., input_{8n-7}\}.$$

3.3.3 The State

Internally, the AES algorithm's operations are performed on a two-dimensional array of bytes called the **State**. The State consists of four rows of bytes, each containing **Nb** bytes, where **Nb** is the block length divided by 32. In the State array denoted by the symbol s, each individual byte has two indices, with its row number r in the range $0 <= r < 4$ and its column number c in the range $0 <= c <$ **Nb**. This allows an individual byte of the State to be referred to as either $s_{r,c}$ or s[r,c]. For this standard, **Nb**=4, i.e., $0 <= c < 4$.

At the start of the Cipher and Inverse Cipher described in Sec. 3.7, the input – the array of bytes $in_0, in_1, \ldots in_{15}$ – is copied into the State array as illustrated in Fig. 3. The Cipher or Inverse Cipher operations are then conducted on this State array, after which its final value is copied to the output – the array of bytes: $out_0, out_1, \ldots out_{15}$.

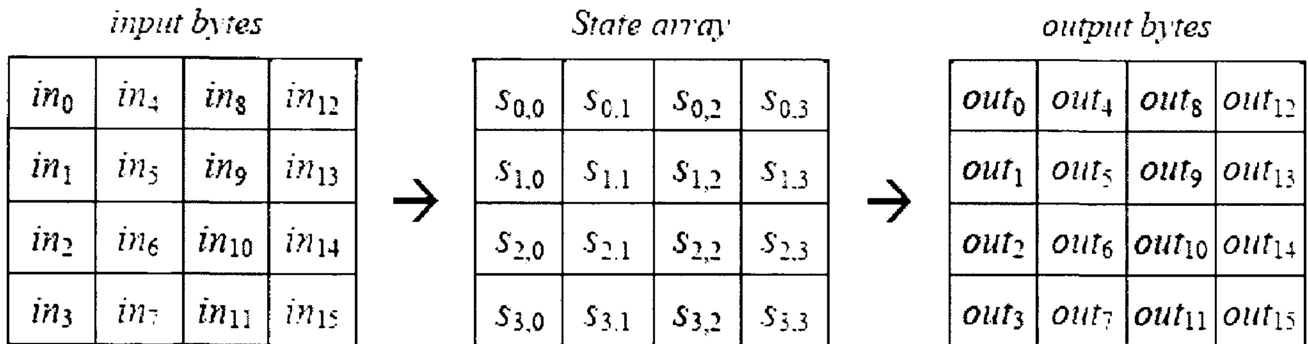


Figure 3.2 State array input and output.

Hence, at the beginning of the Cipher or Inverse Cipher, the input array, *in*, is copied to the State array according to the scheme:

s[r, c] = in[r + 4c] for 0 <= r < 4 and 0 <= c < Nb,

and at the end of the Cipher and Inverse Cipher, the State is copied to the output array out as follows:
out[r + 4c] = s[r, c] for 0 <= r < 4 and 0 <= c < Nb.

3.3.4 The State as an Array of Columns

The four bytes in each column of the State array form 32-bit **words**, where the row number *r* provides an index for the four bytes within each word. The state can hence be interpreted as a one-dimensional array of 32 bit words (columns), $w_0 \ldots w_3$, where

the column number *c* provides an index into this array. Hence, for the example in Fig. 3, the State can be considered as an array of four words, as follows:

$w_0 = s_{0,0}\, s_{1,0}\, s_{2,0}\, s_{3,0}$ $\qquad$ $w_2 = s_{0,2}\, s_{1,2}\, s_{2,2}\, s_{3,2}$

$w_1 = s_{0,1}\, s_{1,1}\, s_{2,1}\, s_{3,1}$ $\qquad$ $w_3 = s_{0,3}\, s_{1,3}\, s_{2,3}\, s_{3,3}$.

3.4 Algorithm Specification

For the AES algorithm**, the length of the input block, the output block and the State is 128 bits.** This is represented by **Nb** = 4, which reflects the number of 32-bit words (number of columns) in the State.

For the AES algorithm**, the length of the Cipher Key, K, is 128, 192, or 256 bits.** The key length is represented by **Nk** = 4, 6, or 8, which reflects the number of 32-bit words (number of columns) in the Cipher Key. [6]

For the AES algorithm, the number of rounds to be performed during the execution of the
algorithm is dependent on the key size. The number of rounds is represented by **Nr**, where **Nr** =10 when **Nk** = 4, **Nr** = 12 when **Nk** = 6, and **Nr** = 14 when **Nk** = 8.

The only Key-Block-Round combinations that conform to this standard are given in Fig. 4. For implementation issues relating to the key length, block size and number of rounds. [5, 6]

	Key Length (*Nk words*)	Block Size (*Nb words*)	Number of Rounds (*Nr*)
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Figure 3.3. Key-Block-Round Combinations.

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.

3.5 Cipher

At the start of the Cipher, the input is copied to the State array using the conventions described in Sec. 3.4. After an initial Round Key addition, the State array is transformed by implementing a round function 10, 12, or 14 times (depending on the key length), with the final round differing slightly from the first **Nr**-1 rounds.

The round function is parameterized using a key schedule that consists of a one-dimensional array of four-byte words derived using the Key Expansion routine.

The Cipher is described in the pseudo code in Fig. 3.4. The individual transformations -

SubBytes(), ShiftRows(), MixColumns(), and AddRoundKey() – process the State and are described in the following subsections. In Fig. 3.4, the array w[] contains the key schedule. As shown in Fig. 3.4, all *Nr* rounds are identical with the exception of the final round, which does not include the MixColumns() transformation.

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
byte state[4,Nb]
state = in
AddRoundKey(state, w[0, Nb-1])
for round = 1 step 1 to Nr–1
SubBytes(state)
ShiftRows(state)
MixColumns(state)
AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
end for
SubBytes(state)
ShiftRows(state)
AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])
out = state
end
```

Figure 3.4. Pseudo Code for the Cipher.

3.5.1 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 (Fig. 3.6), which is invertible, is constructed by composing two transformations:

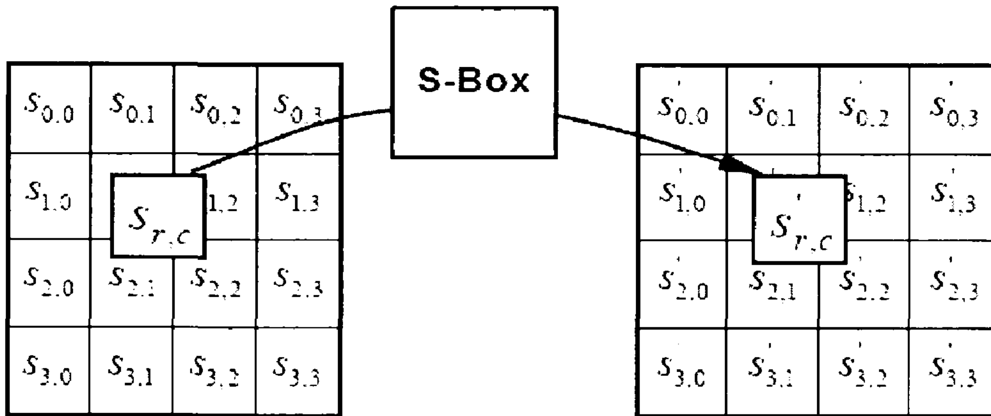


Figure 3.5 SubBytes() applies the S-box to each byte of the State.

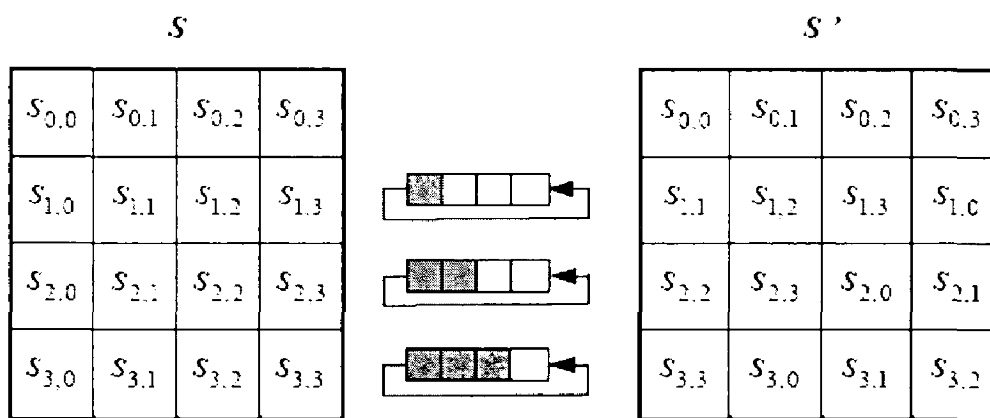
The S-box used in the SubBytes() transformation is presented in hexadecimal form in Fig. 3.6. For example, if s[1,1] = {53}, then the substitution value would be determined by the intersection of the row with index '5' and the column with index '3' in Fig. 3.6. This would result in s'[1,1] having a value of {ed}.

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

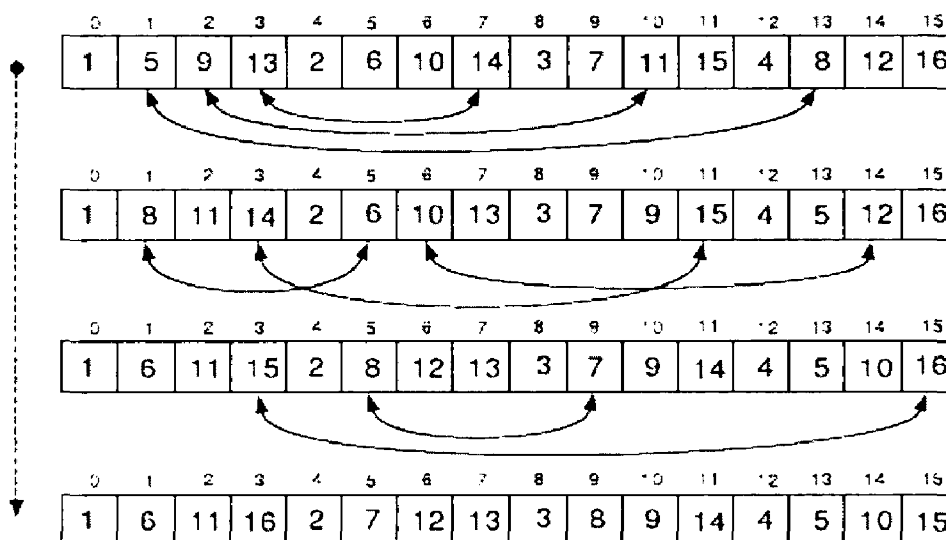
Figure 3.6 S-box: substitution values for the byte xy (in hexadecimal format).

3.5.2 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, $r = 0$, is not shifted. This has the effect of moving bytes to "lower" positions in the row, while the "lowest" bytes wrap around into the "top" of the row.



(a)



(b)

Figure 3.7: The shift of rows (a) in general and (b) in this implementation

3.5.3 MixColumns() Transformation

The MixColumns()transformation operates on the State column-by-column, treating each column as a four-term polynomial.

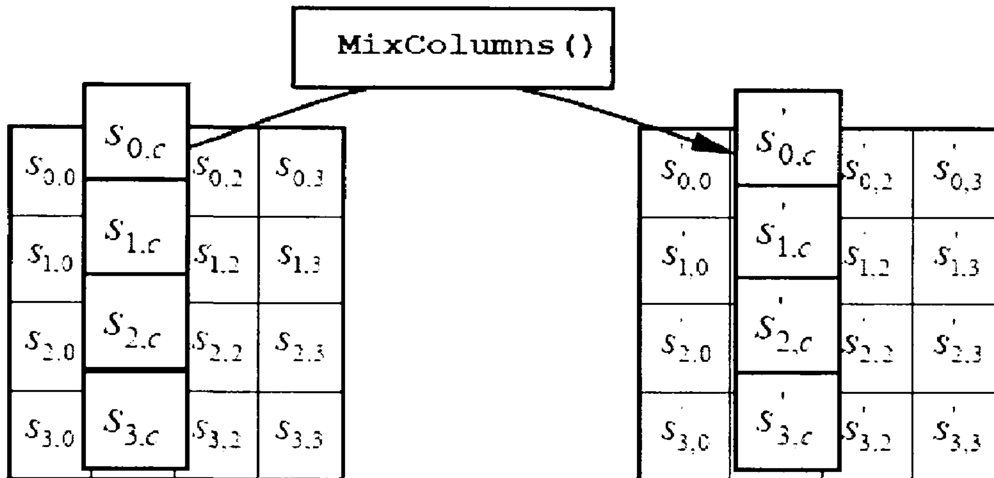


Figure 3.8 MixColumns() operates on the State column-by-column.

3.5.4 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. Those **Nb** words are each added into the columns of the State. In the Cipher, the initial Round Key addition occurs when *round* = 0, prior to the first application of the round function (see Fig. 3.4). The application of the AddRoundKey()transformation to the **Nr** rounds of the Cipher occurs when 1<= *round* <= **Nr**.

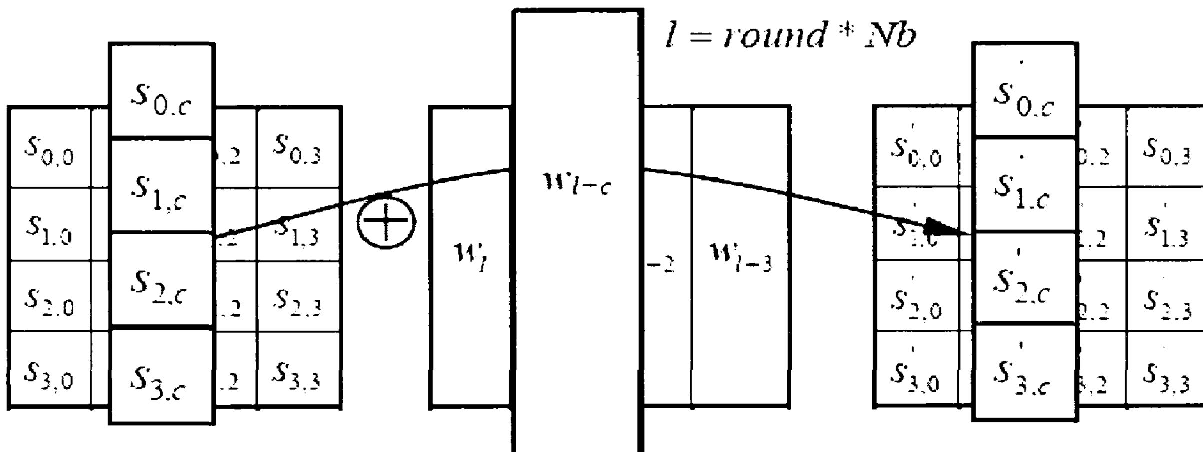


Figure 3.9 AddRoundKey() XORs each column of the State with a word from the key schedule.

3.6 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 [Wi], with i in the range 0 <= i < **Nb**(**Nr** + 1).

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].

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
word temp
i = 0
while (i < Nk)
w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
i = i+1
end while
i = Nk
while (i < Nb * (Nr+1)]
temp = w[i-1]
if (i mod Nk = 0)
temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
else if (Nk > 6 and i mod Nk = 4)
temp = SubWord(temp)
end if
w[i] = w[i-Nk] xor temp
i = i + 1
end while
end
```

Figure 3.10 Pseudo Code for Key Expansion.

3.7 Inverse Cipher

The Cipher transformations in Sec. 3.5 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. [5,6]

```
InvCipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
byte state[4,Nb]
state = in
AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1]) // See Sec. 5.1.4
for round = Nr-1 step -1 downto 1
InvShiftRows(state) // See Sec. 5.3.1
InvSubBytes(state) // See Sec. 5.3.2
AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
InvMixColumns(state) // See Sec. 5.3.3
end for
InvShiftRows(state)
InvSubBytes(state)
AddRoundKey(state, w[0, Nb-1])
out = state
end
```

Figure 3.11 Pseudo Code for the Inverse Cipher.

3.7.1 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, $r = 0$, 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.

3.7.2 InvSubBytes() Transformation

InvSubBytes() is the inverse of the byte substitution transformation, in which the inverse Sbox is applied to each byte of the State.

3.7.3 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

3.7.4 Inverse of the AddRoundKey() Transformation

AddRoundKey() is its own inverse, since it only involves an application of the XOR operation.

3.7.5 Equivalent Inverse Cipher

In the straightforward Inverse Cipher presented in Fig. 3.12, 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. [6]

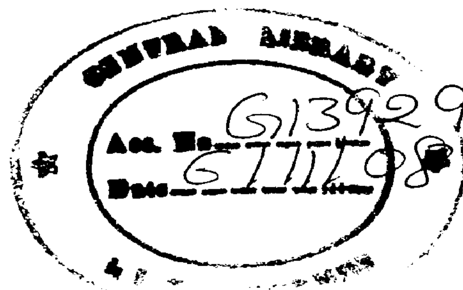
The two properties that allow for this Equivalent Inverse Cipher are as follows:

1. The SubBytes() and ShiftRows() transformations commute; that is, a SubBytes() transformation immediately followed by a ShiftRows() transformation is equivalent to a ShiftRows() transformation immediately followed buy a SubBytes() transformation. The same is true for their inverses, InvSubBytes() and InvShiftRows.

2. The column mixing operations - MixColumns() and InvMixColumns() – are linear with respect to the column input, which means

InvMixColumns(state XOR Round Key)
= InvMixColumns(state) XOR InvMixColumns(Round Key).

These properties allow the order of InvSubBytes() and InvShiftRows() transformations to be reversed. The order of the AddRoundKey() and InvMixColumns() transformations can also be reversed, provided that the columns (words) of the decryption key schedule are modified using the InvMixColumns() transformation.



```
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, (Nr+1)*Nb-1])
for round = Nr-1 step -1 downto 1
InvSubBytes(state)
InvShiftRows(state)
InvMixColumns(state)
AddRoundKey(state, dw[round*Nb, (round+1)*Nb-1])
end for
InvSubBytes(state)
InvShiftRows(state)
AddRoundKey(state, dw[0, Nb-1])
out = state
end

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

Figure 3.12: Pseudo Code for the Equivalent Inverse Cipher.

CHAPTER 4
IMPLEMENTATION

I am going use the name XMODE for this new variant of the CBC mode for our convenience. It takes only one key, K (k bits) of a block cipher E. The key length, k bits, is the minimum because the underlying block cipher must have a k-bit key K anyway.

XMODE is a simple variant of the CBC MAC (Cipher **B**lock **C**haining **M**essage **A**uthentication **C**ode). It allows and is secure for messages of any bit length (while the CBC MAC is only secure on messages of one fixed length, and the length must be a multiple of the block length). Also, the efficiency of XMODE is highly optimized. It is almost as efficient as the CBC MAC.

4.1 Pre-processing

The following steps can be done without the message.

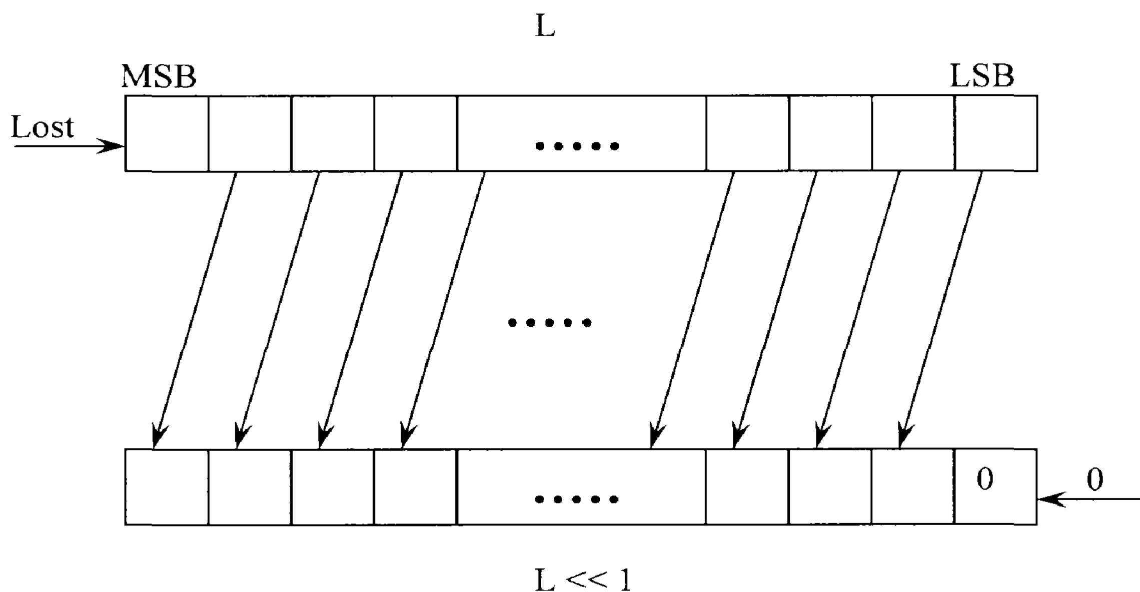


Fig. 4.1 Left shifting by one bit

1. First, encrypt n-bit 0 (denoted by 0^n) to compute L . That is, let L be $E(K, 0^n)$.
2. Check if the most significant bit of L is 0 .
   - If it is, let $L.u$ be $L<<1$, where $L<<1$ denotes a shift in which bits increase in significance with the most significant bit being lost and a zero coming into the least significant bit. See the figure.
   - Otherwise, let $L.u$ be $(L<<1)$ xor $Constant$, where $Constant$ is the n-bit constant. If $n=128$, then $Constant$ is 0x00000000000000000000000000000087, and if $n=64$, then $Constant$ is 0x000000000000001b, where bits are presented as hexadecimal values with their most significant bits to the left.
3. Check if the most significant bit of $L.u$ is 0 .
   - If it is, let $L.u^2$ be $(L.u)<<1$.
   - Otherwise, let $L.u^2$ be $((L.u)<<1)$ xor $Constant$, where $Constant$ is the same as above.
4. Save $L.u$ and $L.u^2$.

4.2 Tag-generation

Let M be the message. Break M into blocks $M[1], M[2], \ldots, M[m]$, where each $M[i]$ $(i = 1, \ldots, m-1)$ is n bits. The last message block $M[m]$ may have fewer than n bits (but it has 0 bits only if the message M is empty)

1. Let $Y[0]$ be 0^n .
2. For $i = 1$ to $m-1$ do : let $Y[i]$ be $E(K, M[i]$ xor $Y[i-1])$.
3. Check if the bit length of the last message block $M[m]$ is n bits.

   - If it is, let $X[m]$ be $M[m]$ xor $Y[m-1]$ xor $L.u$.

   Otherwise,

   - let $M[m]$ be $M[m]\ 1\ 0^{(n-1-(\text{bit length of } M[m]))}$. That is, append a 1 and then append the minimum number of 0s, so that the total length becomes n bits. Let $X[m]$ be $M[m]$ xor $Y[m-1]$ xor $L.u^2$.

4. Let T be $E(K, X[m])$.
5. Let Tag be the t-bit truncation of T .
6. Return Tag.

Note that if the message length is a positive multiple of n , then $L.u$ is used. Otherwise 10^{i} padding and $L.u^{2}$ are used. If the message is an empty string, then you have to append 10^{n-1} and use $L.u^{2}$.

Here is the algorithmical description in pseudocode.

Algorithm XMODE(K,M)

1. L ← E(K, 0_n)
2. **if** msb(L) = 0 **then** L · u ← L << 1
   else L · u ← (L << 1) ex-or Constant
   if msb(L · u) = 0 **then** L · u_2 ← (L · u) << 1
   else L · u_2 ← ((L · u) << 1) ex-or Constant
3. Y [0] ← 0_n
   Break M into blocks M[1],M[2], . . . , M[m]
   /* |M[i]| = n for i = 1, . . .,m − 1, and |M[m]| ≤ n */
4. **for** i ← 1 **to** m − 1 **do**
   Y [i] ← E(K,M[i] ex-or Y [i − 1])
5. **if** |M[m]| = n **then** X[m] ← M[m] ex-or Y [m − 1] ex-or L · u
   else X[m] ← (M[m]$10_{n-1-|M[m]|}$) ex-or Y [m − 1] ex-or L · u_2
6. T ← E(K,X[m])
7. Tag ← t-bit truncation of T
   return Tag

4.3 Tag-verification

Suppose that you have received a message-tag pair *(M, Tag ')* . To check if *(M, Tag ')* is authentic, first, compute the tag *Tag* for the message *M* using the above Tag-generation and your own secret key. If *Tag ' = Tag* then *M* is authentic. Otherwise, *M* is unauthentic.

Case $|M| = mn$ for some $m \geq 1$. In this case, $M = M[1], M[2], \ldots, M[m]$ and $|M[m]| = n$.
Fig

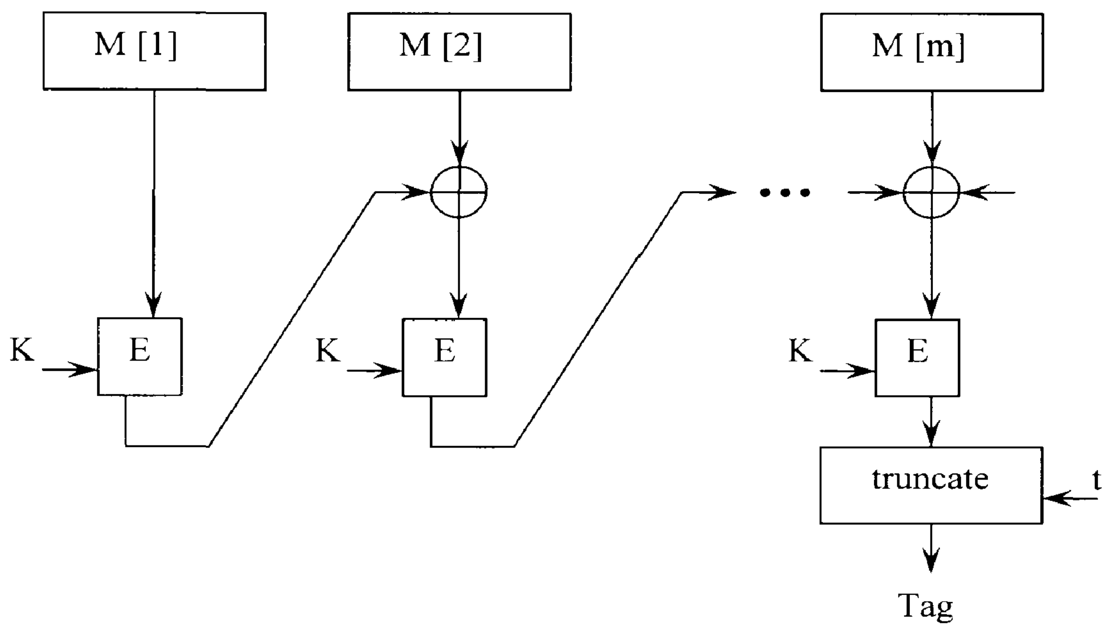


Fig. 4.2 Tag generation

CHAPTER 5
RESULTS AND DISCUSSION

5.1 Results

- "K len." denotes the key length.
- "#K sche." denotes the number of block cipher key schedulings. For RMAC, it requires one block cipher key scheduling each time generating a tag.
- "#M" denotes the number messages which the sender has MACed.
- "#E invo." denotes the number of block cipher invocations to generate a tag for a message M, assuming $|M| > 0$.
- "#E pre." denotes the number of block cipher invocations during the pre-processing time. These block cipher invocations can be done without the message.
- "+kst" means that the key separation technique is used. OMAC does not need the key separation technique since its key length is optimal in its own form.
- For RMAC2, we assume that AES128 is used to compute the CBC MAC tag, and AES256 is used to encrypt it.

Table 5.1 Efficiency Comparison with AES128.

Name	K len.	#K sche.	#E invo.	#E pre.
RMAC1[AES128]	256	1 + #M	1 + (M+1)/128	0
RMAC2[AES128]	384	1+ #M	1 + (M/128)	0
EMAC[AES128]	256	2	1 + (M+1)/128	0
XCBC[AES128]	384	1	M/128	0
TMAC[AES128]	256	1	M/128	0
XMODE[AES128]	128	1	M/128	1

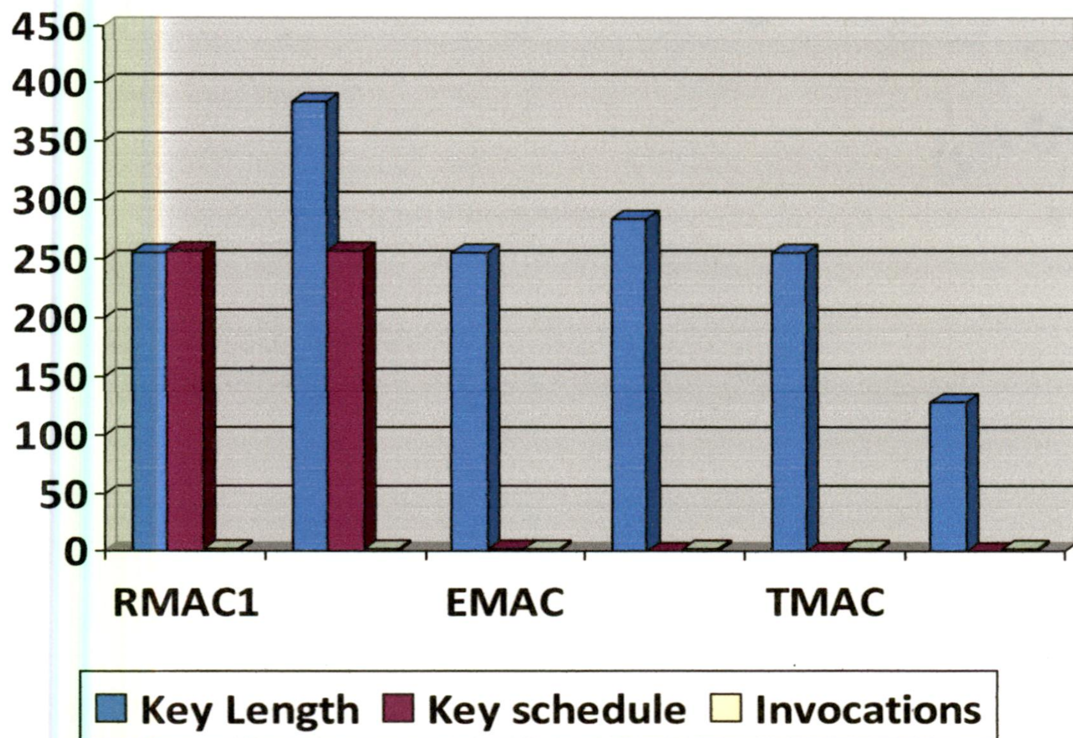


Figure 5.1: Efficiency Comparison with AES128

Table 5.2 Efficiency Comparison with AES192.

Name	K len.	#K sche.	#E invo.	#E pre.
RMAC1[AES192]	384	1 + #M	1+ (M+1)/128	0
EMAC[AES192]	384	2	1 +(M+1)/128	0
XCBC[AES192]	448	1	M/128	0
TMAC[AES192]	320	1	M/128	0
XMODE[AES192]	192	1	M/128	1

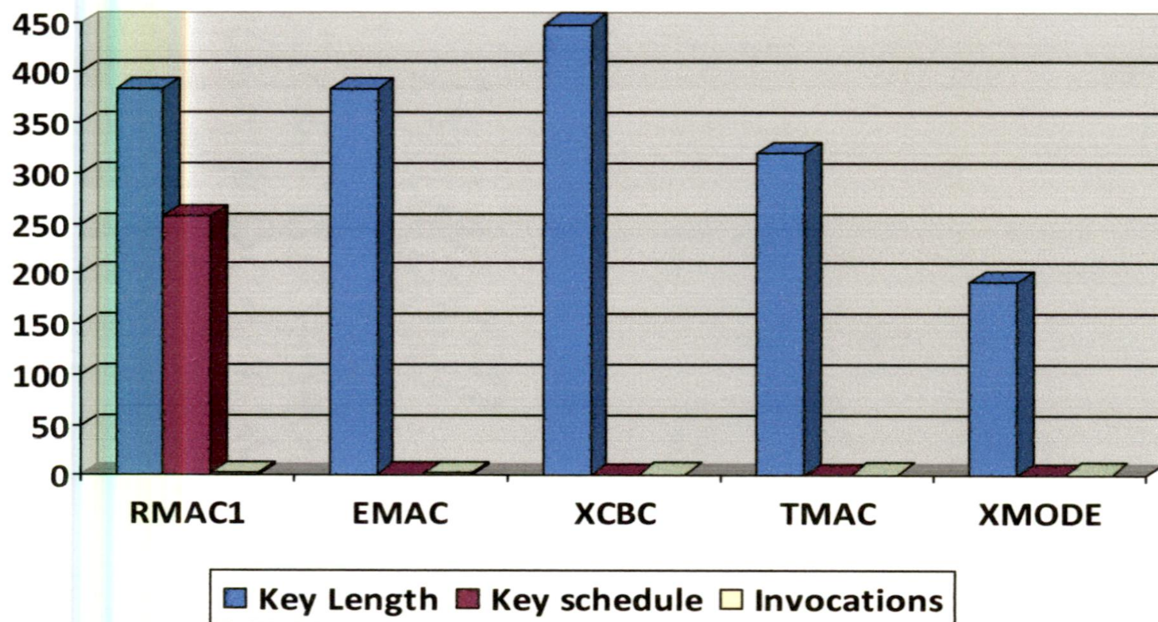


Figure 5.2: Efficiency Comparison with AES192

Table 5.3 Efficiency Comparison with AES256

Name	K len.	#K sche.	#E invo.	#E pre.
RMAC1[AES256]	512	1 + #M	1+ (M+1)/128	0
EMAC[AES256]	512	2	1 +(M+1)/128	0
XCBC[AES256]	512	1	M/128	0
TMAC[AES256]	384	1	M/128	0
XMODE[AES256]	256	1	M/128	1

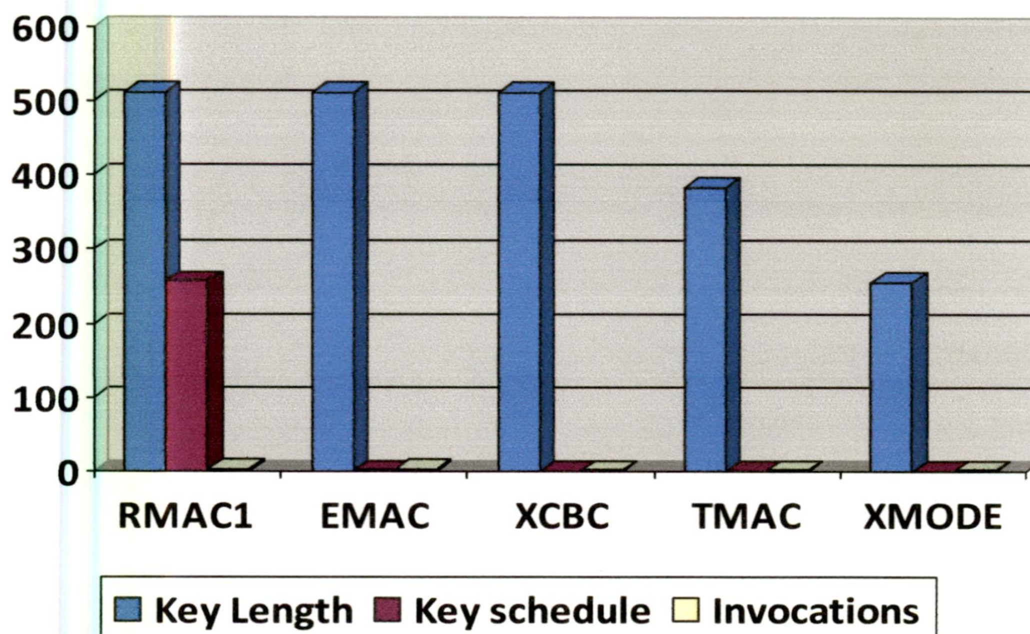


Figure 5.3: Efficiency Comparison with AES256

Table 5.4 Efficiency Comparison with AES128 and Key Separation Technique.

Name	K len.	#K sche.	#E invo.	#E pre.
RMAC1[AES128]	128	2 + #M	1 + (M+1)/128	0
RMAC2[AES128]	128	2 + #M	1 + (M/128)	0
EMAC[AES128]	128	3	1 + (M+1)/128	0
XCBC[AES128]	128	2	M/128	0
TMAC[AES128]	128	2	M/128	0
XMODE[AES128]	128	1	M/128	1

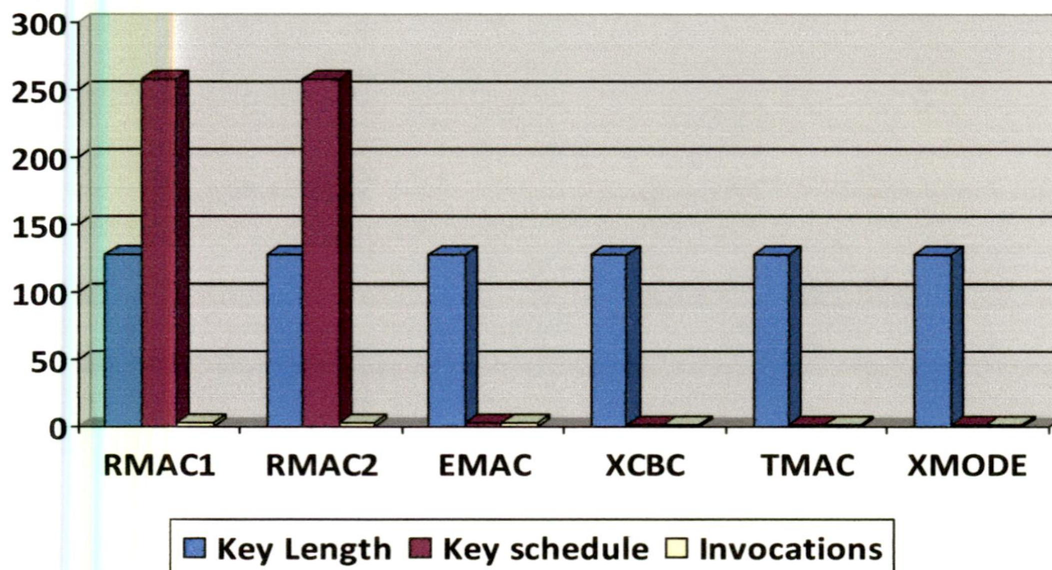


Figure 5.4: Efficiency Comparison with AES128 and Key Separation Technique

Table 5.5 Efficiency Comparison with AES192 and Key Separation Technique..

Name	K len.	#K sche.	#E invo.	#E pre.
RMAC1[AES192]	192	2 + #M	1+ (M+1)/128	3
EMAC[AES192]	192	3	1 +(M+1)/128	3
XCBC[AES192]	192	2	M/128	4
TMAC[AES192]	192	2	M/128	3
XMODE[AES192]	192	1	M/128	1

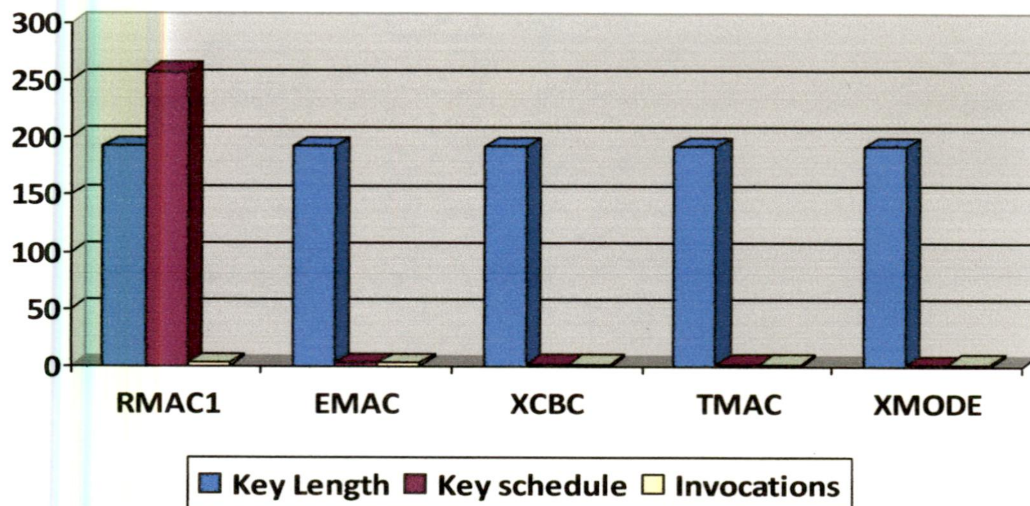


Figure 5.5: Efficiency Comparison with AES192 and Key Separation Technique

Table 5.6 Efficiency Comparison with AES256 and Key Separation Technique.

Name	K len.	#K sche.	#E invo.	#E pre.
RMAC1[AES256]	256	2 + #M	1+ (M+1)/128	4
EMAC[AES256]	256	3	1 +(M+1)/128	4
XCBC[AES256]	256	2	M/128	4
TMAC[AES256]	256	2	M/128	3
XMODE[AES256]	256	1	M/128	1

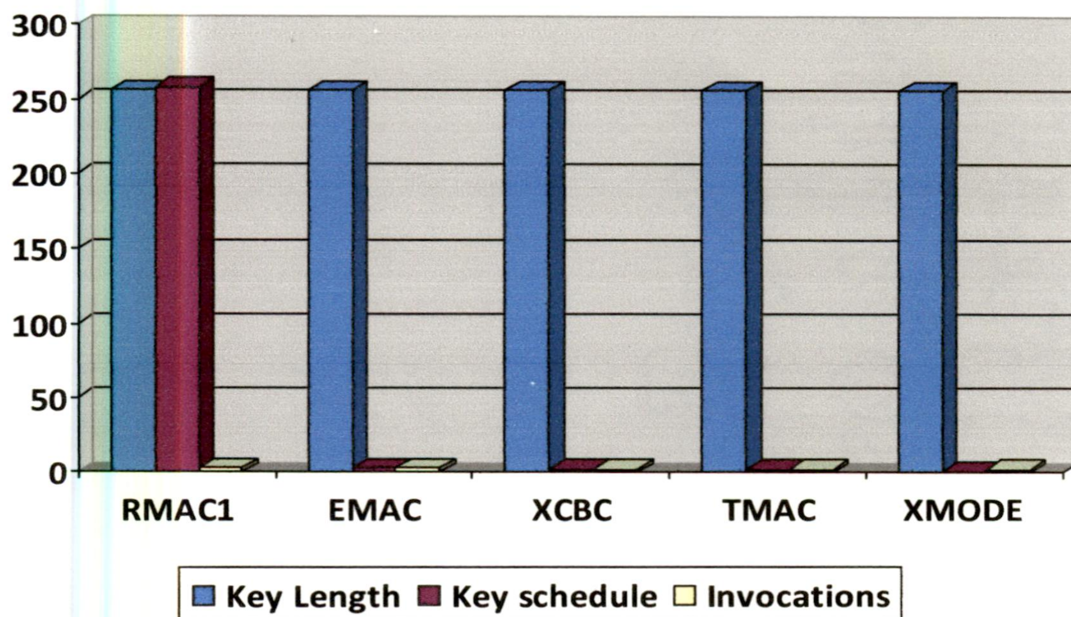


Figure 5.6: Efficiency Comparison with AES256 and Key Separation Technique

5.2 Discussions

None of RMAC. EMAC, XCBC, TMAC and OMAC is optimal in all efficiency measures: "K len.", "#K sche.", "#E invo." and "#E pre." There is a tradeoff among the above four measures.

Key length: In Tables 5.1–5.3, XMODE gives the best performance. It shows that XMODE is as secure as EMAC, XCBC, and TMAC despite of its optimal key length. In Tables 5.4–5.6, the key lengths of RMAC, EMAC, XCBC and TMAC can be reduced to the optimal length. But the cost appears in the number of key schedulings and the number of block cipher invocations during the pre-processing time.

Number of key schedulings: RMAC requires one block cipher key scheduling each time generating a tag. In Tables 5.1–5.3, XCBC, TMAC and XMODE give the best performance, while EMAC requires two block cipher key schedulings. In Tables 5.4–5.6, it is obvious that XMODE gives the best performance.

Number of block cipher invocations: In Tables 5.1–5.6, RMAC and EMAC requires one or two extra block cipher invocations compared to XCBC, TMAC and XMODE.

This overhead is significant for short messages.

Number of block cipher invocations during the pre-processing time: In Tables 5.1–5.3, only XMODE requires one block cipher invocation. But this is not very significant since:

- It can be done in an idle time, and
- It is performed infrequently compared to MAC generation. Thus one or two block cipher invocations to generate a tag in RMAC and EMAC is much more significant since it is performed on each message.

In XMODE, the gain for this cost is its optimal key length, which completely eliminates the need for the key separation technique. We believe this is a very reasonable and desirable tradeoff since:

- key separation technique is a very error-prone process in practice, and
- key separation technique is used in many environment, but if it is used, then other MACs have a significant key setup cost compared to XMODE.

In fact, the performance of XMODE is far better than RMAC, EMAC, XCBC and TMAC in Tables 5.4–5.6.

CHAPTER 6
CONCLUSION

Conclusion

The proposed mode allows and is secure for messages of any bit length (while the CBC MAC is only secure on messages of one fixed length, and the length must be a multiple of the block length). Also, the efficiency of this mode is highly optimized. It is almost as efficient as the CBC MAC.

Proposed mode gives the best performance with AES128, AES 192 and AES 256. It is as secure as EMAC, XCBC, and TMAC despite of its optimal key length. RMAC and EMAC require one or two extra block cipher invocations compared to XCBC, TMAC and proposed mode. It saves the key length, which makes the security proof of proposed mode substantially harder than those of XCBC.

REFERENCES

[1] "AES Algorithm (FIPS-197) Advanced Encryption Standard", http://www.vocal.com , (Last access date October 24, 2007).

[2] J. Daemen and V. Rijmen, The block cipher Rijndael, Smart Card research and Applications, LNCS 1820, Springer-Verlag, pp. 288-296.

[3] A. Lee, Guideline for Implementing Cryptography in the Federal Government, National Institute of Standards and Technology, NIST Special Publication, November 1999, pp. 800-21.

[4] "Implementation Experience with AES Candidate Algorithms", http://www.seven77.demon.co.uk/aes.htm, (Last access date 25-11-2007).

[5] "Advanced Encryption Standard (AES).", U.S. DoC/NIST, FIPS Publication 197, November 26, 2001.

[6] "Data Encryption Standard (DES).", U.S. DoC/NIST, FIPS Publication 46-3, October 25, 2004.

[7] K.G. Paterson and A. Yau, "Padding Oracle Attacks on the ISO CBC Mode Encryption Standard", Topics in Cryptology - CT-RSA '04, LNCS, Vol. 2964, pp. 305-323, Springer-Verlag, 2004.

[8] S. Vaudenay, "Security Flaws Induced by CBC Padding – Applications to SSL, IPSEC, WTLS", Advances in Cryptology - Eurocrypt '02, LNCS, Vol. 2332, pp. 534-545, Springer-Verlag, 2002.

[9] H. Wu, "Related-Cipher Attacks", ICICS '02, LNCS, Vol. 2513, pp. 447-455, Springer-Verlag, 2002.

[10] E. Biham, J. Seberry, "Tweaking the IV Setup of the Py Family of Ciphers – The Ciphers Tpy, TPypy, and TPy6," Published on the author's webpage at http://www.cs.technion.ac.il/ biham/, January 25, 2007.

[11] S. Paul, B. Preneel "On the (In)security of Stream Ciphers Based on Arrays and Modular Addition," siacrypt 2006 (X. Lai and K. Chen, eds.), vol. 4284 of LNCS, pp. 69-83, Springer-Verlag, 2006.

[12] P. Rogaway (submitter) and M. Bellare, J. Black, and T. Krovetz (auxiliary submitters). OCB mode. Contribution to NIST. Cryptology ePrint archive, report 2001/26, Apr 1, 2001, revised Apr 18, 2001. ePrint.iacr.org and csrc.nist.gov/encryption/modes/proposedmodes.

[13] A. Joux, "Cryptanalysis of the EMD Mode of Operation", Advances in Cryptology - Eurocrypt '03, LNCS, Vol. 2656, pp. 1-16, Springer-Verlag, 2003.

[14] M. Bellare, P. Rogaway and D. Wagner. "The EAX mode of operation", University of California Postprints, Paper 1218, 2004.

[15] "Block cipher modes of operation", **http://en.wikipedia.org/wiki/Block cipher modes of operation**, (Last access date May 24, 2008).