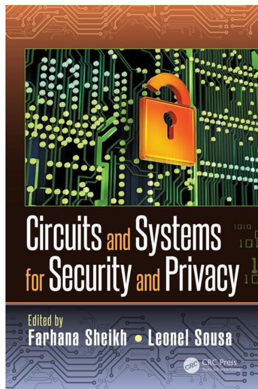
This article was downloaded by: *10.2.98.160*
On: 27 Oct 2020
Access details: *subscription number*
Publisher: *CRC Press*
Informa Ltd Registered in England and Wales Registered Number: 1072954 Registered office: 5 Howick Place, London SW1P 1WG, UK



Circuits and Systems for Security and Privacy

Farhana Sheikh, Leonel Sousa, Krzysztof Iniewski

Block Ciphers

Publication details
https://test.routledgehandbooks.com/doi/10.1201/b19499-4
Deniz Toz, Josep Balasch, Farhana Sheikh
Published online on: 24 May 2016

How to cite :- Deniz Toz, Josep Balasch, Farhana Sheikh. 24 May 2016, *Block Ciphers from:* Circuits and Systems for Security and Privacy CRC Press
Accessed on: 27 Oct 2020
https://test.routledgehandbooks.com/doi/10.1201/b19499-4

PLEASE SCROLL DOWN FOR DOCUMENT

Full terms and conditions of use: https://test.routledgehandbooks.com/legal-notices/terms

This Document PDF may be used for research, teaching and private study purposes. Any substantial or systematic reproductions, re-distribution, re-selling, loan or sub-licensing, systematic supply or distribution in any form to anyone is expressly forbidden.

The publisher does not give any warranty express or implied or make any representation that the contents will be complete or accurate or up to date. The publisher shall not be liable for an loss, actions, claims, proceedings, demand or costs or damages whatsoever or howsoever caused arising directly or indirectly in connection with or arising out of the use of this material.

3

Block Ciphers

Deniz Toz
K. U. Leuven

Josep Balasch
K. U. Leuven

Farhana Sheikh
Intel Corporation

CONTENTS

3.1 Introduction 64
3.2 Block Cipher Definition 64
3.3 Historical Background 65
3.4 DES 66
3.4.1 The Round Function 66
3.4.2 The Key Schedule 66
3.4.3 Decryption 68
3.5 AES 68
3.5.1 The Round Transformation 68
3.5.1.1 The `SubBytes` step 68
3.5.1.2 The `ShiftRows` step 69
3.5.1.3 The `MixColumns` step 69
3.5.1.4 The `AddRoundKey` step 70
3.5.2 The Key Schedule 70
3.5.3 Decryption 70
3.6 Modes of Operation 71
3.7 Composite Field Representation of S-box 73
3.8 Software Implementations of AES 74
3.8.1 8-Bit Processors 75
3.8.2 32-Bit Processors 76
3.8.3 64-Bit Processors 76
3.9 Hardware Implementations of AES 77
3.9.1 AES SBox Optimization for Hardware 77
3.9.2 AES for Microprocessors 78
3.9.3 AES for Ultra-Low Power Mobile Platforms 78
3.10 Future Directions 79

© 2016 by Taylor & Francis Group, LLC

Downloaded By: 10.2.98.160 At: 00:35 27 Oct 2020; For: 9781482236897, chapter3, 10.1201/b19499-4

3.1 Introduction

Public and private key cryptography can provide secure and reliable transmission of digital information over either a wired or wireless channel. In order to operate efficiently in both wired and wireless real-time environments, cryptography systems must deliver high throughput. For wireless communication, low-power is an additional necessary requirement. Specialized hardware to meet these needs as (wired or wireless) general-purpose programmable devices with software implementations cannot deliver the required energy-efficiency and throughput. This chapter presents an introduction to block ciphers typically used in private key cryptography systems and the most common block cipher used today is the Advanced Encryption Standard also known as AES. Its predecessor was the Data Encryption Standard. The next section presents the mathematical definition of a block cipher followed by a historical review of block ciphers. The remainder of the chapter then focuses on AES and its implementations in both hardware and software.

3.2 Block Cipher Definition

A block cipher is a mathematical function that acts on fixed-length input values and returns output values of the same length by using a secret key. The process of transforming the input, known as *plaintext*, is called *encryption*. The resulting output is called *ciphertext* and is unaccessible to anyone without the knowledge of the secret key. The original message can be revealed by an inverse process called *decryption*. More formally we define a block cipher as follows:

Definition 3.2.1 *A block cipher is a function* $E : \{0,1\}^n \times \{0,1\}^k \rightarrow \{0,1\}^n$, $E(P,K) = C$ *such that for each* $K \in \{0,1\}^k$, E *is a bijection (i.e., invertible mapping) from* $\{0,1\}^n$ *to* $\{0,1\}^n$, *where* P *is the plaintext,* C *is the ciphertext, and* K *is the secret key.*

The plaintext is partitioned into blocks of fixed size n and each block is encrypted separately using the same secret key. The ciphertext is then obtained by combining the outputs of all encryptions. The exact method by which the outputs are combined are specified by the *mode of operation* of the block cipher. A brief summary of these modes will be provided later in this chapter.

© 2016 by Taylor & Francis Group, LLC

Downloaded By: 10.2.98.160 At: 00:35 27 Oct 2020; For: 9781482236897, chapter3, 10.1201/b19499-4

Most block ciphers use an iterative round function (based on Shannon's product cipher [387]) as the building block. The main idea is to combine two or more simple operations such as modular arithmetic, substitution, and permutation to obtain a more secure cipher (than either of its components). Almost all of the algorithms used today are based on this concept. Each round function is key-dependent; hence, the initial key (known as the *master key*) is expanded into the round keys by the key scheduling algorithm.

3.3 Historical Background

Even though computers were initially only for governmental and military use, in the 1960s they became affordable and powerful enough also for the private sector. Consequently, the need of a common system for communicating with the other companies in addition to internal communication arose. Lucifer [120] was developed by IBM to fulfill this need.

With the beginning of the information age in the 1970s, the exchange of digital information became an essential part of our society. In 1973, the National Bureau of Standards (NBS) made a call for a candidate symmetric-key encryption algorithm for the protection of sensitive but unclassified information. Unfortunately, none of the proposals were found viable and a second call was issued in 1974. After the evaluation phase, the submission of IBM which was heavily influenced by Lucifer, was chosen to become the Data Encryption Standard (DES) after some modifications.

At the end of 1980s and the beginning of 1990s new block ciphers were designed as an alternative to DES. Some examples include RC5, IDEA, FEAL, Blowfish, and CAST. Meanwhile the cryptographers made a great effort in analyzing the security of DES. Differential cryptanalysis [44] and linear cryptanalysis [278] which are the core of many other cryptanalysis techniques that are used today, were introduced in this period. The DESCHALL Project, which consisted of thousands of volunteers connected over the Internet, was the first to break DES (by using exhaustive key search) in public in 1997. Only two years later, it was possible to perform an exhaustive key search for DES in less than a day.

Obviously the short key-length of DES was no longer sufficient for sensitive applications. Moreover, being initially designed for hardware performance, DES was not as efficient in software as the new block ciphers. In 1997, NIST (National Institute of Standards and Technology) made a public call for a new algorithm "capable of protecting sensitive government information well into the next century" [306] to replace the DES.

As a result, fifteen submissions were submitted to the competition and after two years evaluation five of them were chosen as finalists. Finally in 2001 Rijndael [94], designed by Rijmen and Daemen, was chosen as the Advanced

© 2016 by Taylor & Francis Group, LLC

Downloaded By: 10.2.98.160 At: 00:35 27 Oct 2020; For: 9781482236897, chapter3, 10.1201/b19499-4

Encryption Standard (AES). Unsurprisingly, in the past 10 years many attacks have been published against AES, yet none of them is considered a practical threat to its security. All the attacks were either against reduced round AES or they worked under some special conditions until the recent work of [50] which reduces the complexity of exhaustive search by a factor of four.

3.4 DES

DES [305] is a block cipher designed by IBM. It was chosen as the Data Encryption Standard in 1977 by the National Bureau of Standards (NBS) for the protection of sensitive, unclassified governmental data. DES accepts a 64-bit plaintext P and a 128-bit user key as inputs and is composed of 16 rounds. The bits of the input block are first shuffled by using an initial permutation (IP), and then the permuted input is divided into two branches L_0 and R_0 where each branch has 32 bits. The two branches are updated by using a round function f as follows:

$$\begin{aligned} L_i &= R_{i-1} \\ R_i &= L_{i-1} \oplus f(R_{i-1}, RK_i) \end{aligned}$$

Here, RK_i is the round key generated from the 56-bit secret key by the key schedule. The number of rounds for DES is 16 and in the last round the swap operation is omitted. Finally, the ciphertext value (C) obtained after the inverse initial permutation (IP^{-1}).

3.4.1 The Round Function

In each round, the 32-bit input block is first expanded to 48 bits by a key expansion E. The expansion permutation simply duplicates half of the bits as follows: let the output of the key expansion be considered as eight groups of 6 bits each, then the j-th group is composed of the bits $(4j-4, 4j-3, \ldots 4j+1)$.

The expanded input block is then XORed with the 48-bit round key and the result passes the substitution layer. DES uses eight 6×4-bit S-boxes $S_1, S_2, \ldots, S_3$ for substitution. This step is the only non-linear operation of DES. Finally all words are permuted by the permutation P. The sketch of DES is given in Figure 3.1.

3.4.2 The Key Schedule

The key schedule derives the sixteen 48-bit round keys from the master key. DES has a very simple key schedule: The 56-bit key is divided into two

© 2016 by Taylor & Francis Group, LLC

Downloaded By: 10.2.98.160 At: 00:35 27 Oct 2020; For: 9781482236897, chapter3, 10.1201/b19499-4

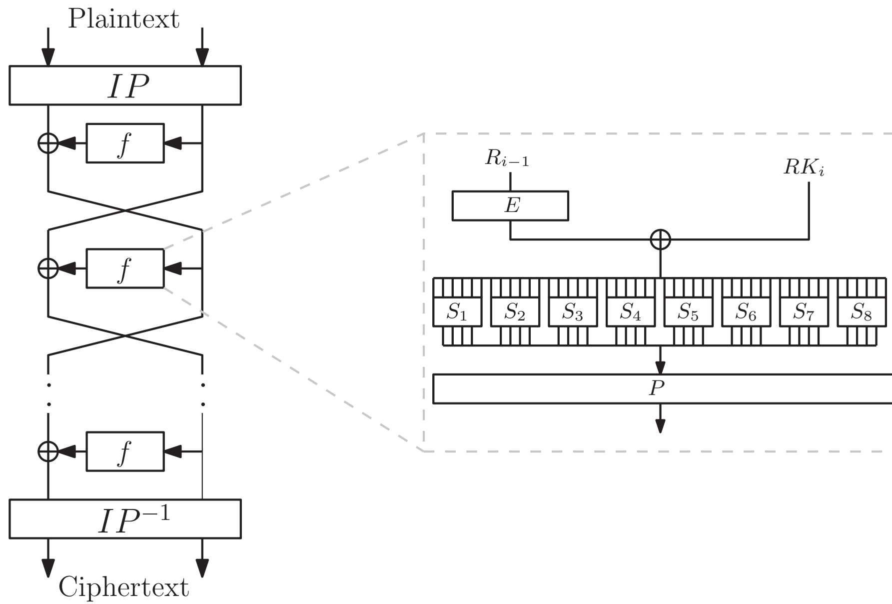


Figure 3.1
The Feistel structure and the round function of DES

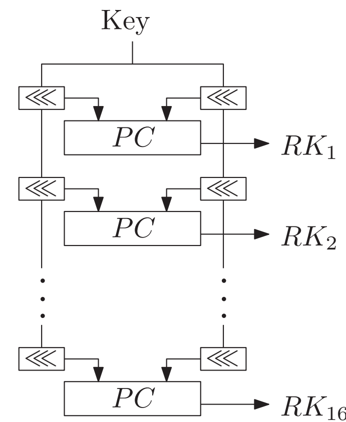


Figure 3.2
The key schedule of DES

branches of 28 bits and each branch is cyclically shifted left by one or two bits (the number of shifts is specified for each round). Then 48-bit subkey is selected by Permuted Choice (PC) and is composed of 24 bits from the left half, and 24 from the right half. The key schedule of DES is depicted in Figure 3.2.

© 2016 by Taylor & Francis Group, LLC

Downloaded By: 10.2.98.160 At: 00:35 27 Oct 2020; For: 9781482236897, chapter3, 10.1201/b19499-4

3.4.3 Decryption

The decryption function is identical to the encryption function except the round keys are used in the reserve order. Permutations IP and IP^{-1} cancel out each other and due to symmetry of the Feistel structure the same inputs enter the round function f. The same key schedule is used for decryption.

3.5 AES

Rijndael [94] is a block cipher designed by Daemen and Rijmen and is a substitution-permutation network following the wide-trail strategy. Both the block length and the key length can be any multiple of 32 bits, with a minimum of 128 bits and a maximum of 256 bits, independently of each other with key size greater than or equal to block size. The 128-bit block variant of Rijndael has been chosen as the Advanced Encryption Standard (AES).

In AES, each data block (plaintext, ciphertext, subkey, or intermediate step) is represented by a 4×4 `state matrix` of bytes. The bytes are considered as elements of the finite field $GF(2^8)$ (i.e., a polynomial with coefficients in $GF(2)$). The irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$ is used to construct $GF(2^8)$; hence, the multiplication is done and the multiplicative inverse is defined accordingly. Here, addition is done in $GF(2)$ which is the XOR operation.

3.5.1 The Round Transformation

The state is initialized with the plaintext block, and then it is then transformed by iterating a round function. The final state gives the ciphertext block. The round function is composed of the four transformations `SubBytes (SB)`, `ShiftRows (SR)`, `MixColumns (MC)`, and `AddRoundKey (AK)`. (See Figure 3.3.) Before the first round, there exists a whitening layer consisting of `AddRoundKey` only, and in the last round the `MixColumns` operation is omitted.

The number of rounds N_r for AES varies with the key length N_k; $N_r = 10$ for 128-bit keys ($N_k = 4$), $N_r = 12$ for 192-bit keys ($N_k = 6$) and $N_r = 14$ for 256-bit keys ($N_k = 8$).

3.5.1.1 The `SubBytes` step

The `SubBytes` step is a non-linear byte substitution (8×8-bit S-box) that acts on every byte of the state. The S-box used in AES is denoted by S_{RD} and is composed of two maps, f and g.

$$S_{RD}(x) = f \circ g(x) = f(g(x))$$

© 2016 by Taylor & Francis Group, LLC

Downloaded By: 10.2.98.160 At: 00:35 27 Oct 2020; For: 9781482236897, chapter3, 10.1201/b19499-4

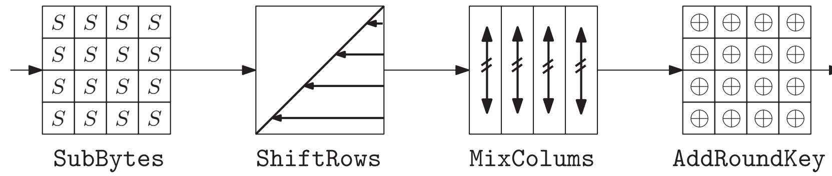


Figure 3.3
The round transformation of AES

Both maps are defined in $GF(2^8)$ and have simple algebraic expressions. Let $x \in GF(2^8)$, then g maps x to its multiplicative inverse x^{-1}. f is an invertible affine transformation given by $f(x) = Ax + b$.

$$f(x) := \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_7 \\ x_6 \\ x_5 \\ x_4 \\ x_3 \\ x_2 \\ x_1 \\ x_0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

3.5.1.2 The `ShiftRows` step

The `ShiftRows (SR)` step is a cyclic shift of bytes in a row that acts individually on each of the last three rows of the state. Row i is shifted i bytes to the left so that the byte at position j is row moves to position $(j - i) \bmod 4$. It aims to provide the optimal diffusion. Similarly, the inverse operation is a cyclic shift of the 3 bottom rows to the right by the same amount.

3.5.1.3 The `MixColumns` step

The `MixColumns` step is a linear transformation that acts independently on every column of the state. The columns of the state are considered as polynomials over $GF(2^8)$ and multiplied with a fixed polynomial $c(x) = \texttt{03} \cdot x^3 + \texttt{01} \cdot x^2 + \texttt{01} \cdot x + \texttt{02}$ modulo $(x^4 + 1)$. Alternatively, the modular multiplication can be written as a matrix multiplication. Let $b(x) = c(x) \cdot a(x)$, then:

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

The inverse operation is similar to `MixColumns`: each column is multiplied with a fixed polynomial $d(x)$ where $c(x) \cdot d(x) \equiv 1 \bmod (x^4 + 1)$. It is computed as $d(x) = \texttt{0B} \cdot x^3 + \texttt{0D} \cdot x^2 + \texttt{09} \cdot x + \texttt{0E}$.

© 2016 by Taylor & Francis Group, LLC

Downloaded By: 10.2.98.160 At: 00:35 27 Oct 2020; For: 9781482236897, chapter3, 10.1201/b19499-4

3.5.1.4 The `AddRoundKey` step

The `AddRoundKey` step is the bitwise exclusive-or (XOR) of the round key with the intermediate state. To invert the `AddRoundKey`, one simply performs the XOR operation with the same round key. Therefore, `AddRoundKey` is its own inverse. The round key of the i-th round is denoted by `ExpandedKey[i]` and is derived from the master key by using a key schedule.

3.5.2 The Key Schedule

The key schedule derives $(N_r + 1)$ 128-bit round keys `ExpandedKey[·]` from the master key. It consists of a linear array of 4-byte words denoted by $W[i]$ for $0 \leq i \leq 4 \cdot (N_r + 1)$. The round key of the i-th round is given by the words $4i$ to $4i + 3$ of W.

There are two versions of the key expansion function, depending on the key size N_k. For both versions, the first N_k words $W[0], W[1], \cdots, W[N_k - 1]$ are directly initialized with the words of the master key. The remaining key words are generated recursively in terms of the previous key words. For $N_k \leq 6$

$$W[i] = \begin{cases} W[i - N_k] \oplus S_{RD}(W[i-1] \lll 8) \oplus RC[i/N_k] & \text{if } i \equiv 0 \text{ mod } N_k \\ W[i - N_k] \oplus W[i-1] & \text{otherwise} \end{cases}$$

For $N_k = 8$, we have

$$W[i] = \begin{cases} W[i - N_k] \oplus S_{RD}(W[i-1] \lll 8) \oplus RC[i/N_k] & \text{if } i \equiv 0 \text{ mod } N \\ W[i - N_k] \oplus S_{RD}(W[i-1]) & \text{if } i \equiv 4 \text{ mod } N \\ W[i - N_k] \oplus W[i-1] & \text{else} \end{cases}$$

In the above equations $\lll$ denotes the rotation of the word to the left, $RC[\cdot]$ are the fixed round constants, and they are independent of N_k. The round key RK_i is given by the words $W[N_b \cdot i]$ to $W[N_b \cdot (i+1)]$.

3.5.3 Decryption

The ciphertext can be decrypted in a straightforward way by using the inverse functions `InvSubBytes`, `InverseShiftRows`, `InverseMixColumns`, and `AddRoundKey` in a reverse order. There exists also an equivalent algorithm for decryption in which the order of the (inverse) steps is the same as in the encryption with a modified key schedule (i.e., `InverseMixColumns` is applied to each `ExpandedKey[i]`). This equivalent algorithm is more convenient for software implementation purposes.

© 2016 by Taylor & Francis Group, LLC

Downloaded By: 10.2.98.160 At: 00:35 27 Oct 2020; For: 9781482236897, chapter3, 10.1201/b19499-4

Downloaded By: 10.2.98.160 At: 00:35 27 Oct 2020; For: 9781482236897, chapter3, 10.1201/b19499-4

3.6 Modes of Operation

In block ciphers, every message block is processed in a similar way called the *mode of operation*. To obtain the ciphertext C, the plaintext P is first padded such that the length of the input is a multiple of the block length. Then it is divided into t blocks such that $P||pad = P_1||P_2||...||P_t$. Up to now many modes of operation have been proposed. Some common modes are given below.

- **Electronic Code Book (ECB).** This is the most straightforward mode of operation in which each plaintext block is encrypted independently by using the same key. Its simplicity and suitability for parallelization are among the advantages of this mode. However, it does not hide patterns in the plaintext, i.e., identical plaintext blocks are encrypted into identical ciphertext blocks. Therefore, the use of this mode is not recommended for cryptographic applications. The scheme is depicted in Figure 3.4.

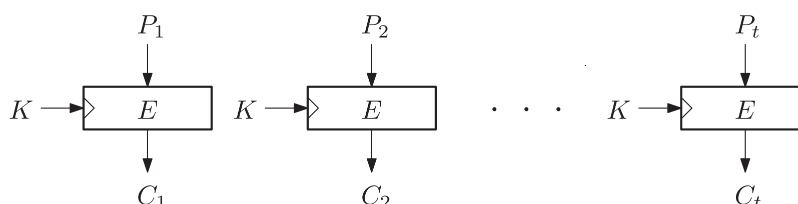


Figure 3.4
Electronic Code Book (ECB) mode of operation

- **Cipher Block Chaining (CBC).** It is the most widely used mode of operation in which each plaintext block is XORed with the previous ciphertext block (an initialization vector (IV) is used for the first plaintext block) before passing through the block cipher. This guarantees that each ciphertext block depends on all previous plaintext blocks, avoiding repetitions and providing randomness. The encryption operation is no longer parallelizable whereas for decryption it is sufficient to have two consecutive ciphertext blocks, and it can still be performed independently. The scheme is depicted in Figure 3.5.

- **Cipher Feedback (CFB).** In CFB mode the previous ciphertext block is encrypted and then XORed with the current plaintext block to obtain the ciphertext block. As in CBC mode, for the first block an initialization vector is used. Hence, this mode of operation converts the block cipher into a self-synchronizing stream cipher. Again, the feedback prevents the parallelization of encryption whereas decryption can be done independently provided that two consecutive blocks are available. This mode was initially

© 2016 by Taylor & Francis Group, LLC

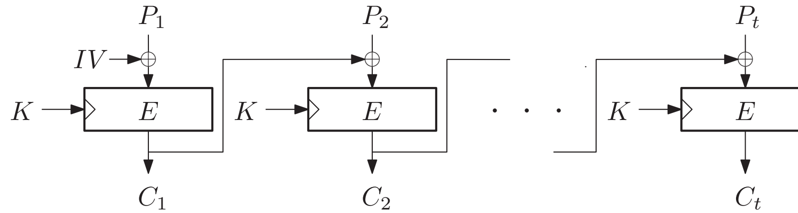


Figure 3.5
Cipher Block Chaining (CBC) mode of operation

designed to encrypt r-bit message blocks and to transmit without delay where $1 \leq r \leq n$ (typically $r = 1$ or $r = 8$). In this case, the r leftmost bits of the output block ($E(S, K)$) is XORed with the plaintext bits and returned as ciphertext bits. The state S is shifted to left by r bits and previous ciphertext bits are inserted before each iteration as feedback. This is depicted in Figure 3.6.

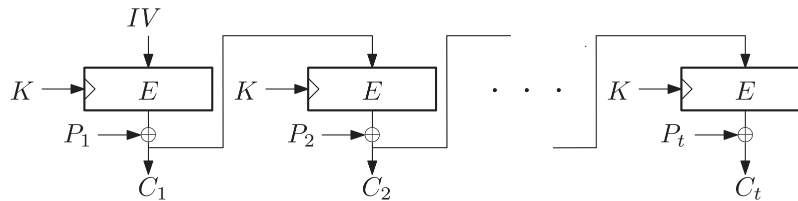


Figure 3.6
Cipher Feedback (CFB) mode of operation

- **Counter (CTR).** In this mode of operation the counter is encrypted and then the result is XORed with the plaintext block to obtain the ciphertext block. The counter incremented before it is used for the next block. Although a function can be used for incrementation, incrementing by one is the easiest and the most common application. The use of nonce (a random or pseudorandom number) is suggested and the same IV and key combination must not be used more than once for protection against attacks. CTR mode allows parallelization. This is depicted in Figure 3.7.

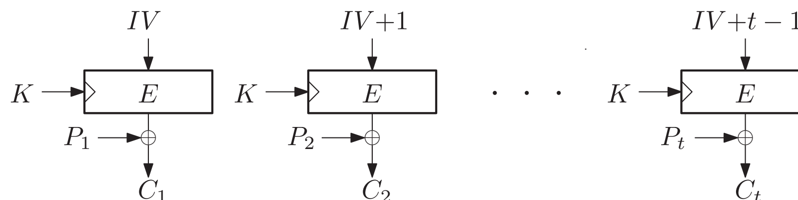


Figure 3.7
Counter (CTR) mode of operation

© 2016 by Taylor & Francis Group, LLC

Downloaded By: 10.2.98.160 At: 00:35 27 Oct 2020; For: 9781482236897, chapter3, 10.1201/b19499-4

3.7 Composite Field Representation of S-box

Galois Field $GF(2^k)$ is a finite field containing 2^k elements and any k linearly independent elements form a basis. For instance, the set

$$B_1 = \{1, \alpha, \alpha^2, \ldots, \alpha^{k-1}\}$$

is a basis for $GF(2^k)$ provided that the elements of B are linearly independent. Each element $a \in GF(2^k)$ can be represented as a polynomial of degree $d < k$ as

$$a = \sum_{i=0}^{k-1} a_i \alpha_i$$

where the coefficients $a_i \in GF(2)$ and B_1 are called the polynomial basis. Various types of bases have been studied extensively; standard, normal, and dual basis can be given among the most common representations. Different basis representations can be used to simplify the implementation of arithmetic operations.

If k is a composite integer (i.e., $k = mn$) then the fields $GF(2^k)$ and $GF((2^m)^n)$ are isomorphic to each other, and it is possible to represent the elements of $GF(2^k)$ as polynomials of degree $d < n$ with coefficients in $GF(2^m)$. In this case, $G(2^k)$ is a *composite field*, and $GF(2^m)$ is called the *ground field*. Now, let B_2 be a polynomial basis for $GF((2^m)^n)$:

$$B_2 = \{1, \beta, \beta^2, \ldots, \beta^{n-1}\}$$

Then $a \in GF(2^k)$ can be represented as:

$$a = \sum_{i=0}^{n-1} a'_i \beta_i$$

where the coefficients $a'_i \in GF(2^m)$. The composite field representation provides relatively efficient implementations when the arithmetic operations (such as multiplication, inversion, and exponentiation) rely on table lookups.

The AES S-box uses the mapping $g : x \to x^{-1}$ where x^{-1} is the multiplicative inverse in $GF(2^8)$. This inversion is the most costly step of the S-box transformation in hardware implementation of AES. The simplest method for finding the inverse is to use a lookup table; however, this method requires $2^m \times m$ bits of memory which increases exponentially with m. An algorithm based on Euclid's algorithm whose area requirement is proportional to m and requires $2m$ cycles per inversion has been presented by Brunner et al. in [58].

The first efficient hardware implementation of the multiplicative inversion in $GF(2^8)$ has been proposed by Rijmen [346] and first implemented

© 2016 by Taylor & Francis Group, LLC

Downloaded By: 10.2.98.160 At: 00:35 27 Oct 2020; For: 9781482236897, chapter3, 10.1201/b19499-4

by Rudra et al. [358] and Wolkerstorfer et al. [455]. The main idea is to decompose the elements of $GF(2^8)$ into linear polynomials over the subfield $GF(2^4)$. In this case, an element $a \in GF(2^8)$ can be represented as:

$$a \cong a_1 x + a_0$$

where $a_1, a_0 \in GF(2^4)$. The multiplication is performed modulo an irreducible polynomial with degree two to ensure that the result is a linear polynomial. Let the irreducible polynomial be denoted as $x^2 + \alpha x + \beta$, then the multiplicative inverse of a can be computed as:

$$a^{-1} \cong a_1(a_1^2\beta + a_1 a_0 \alpha + a_0^2)^{-1} x + (a_0 + a_1\alpha)(a_1^2\beta + a_1 a_0 \alpha + a_0^2)^{-1}$$

As a result, the inversion in $GF(2^8)$ can be expressed in terms of multiplications, squarings, additions, and inversion in $GF(2^4)$. Moreover, the elements of $GF(2^4)$ can be further decomposed into polynomials over the $GF(2^2)$ before implementing the final operations in $GF(2)$.

$$GF(2^8) \cong GF((2^4)^2) \cong GF(((2^2)^2)^2) \cong GF((((2)^2)^2)^2)$$

This method is known as *tower field decomposition*. Satoh et al. [370] and Mentens et al. [286] follow this approach in their implementations. In all the above implementations the field elements are represented by using polynomial basis.

3.8 Software Implementations of AES

Due to its structural and algebraic simplicity, the AES algorithm is well suited for implementations on a wide variety of processors. Generic software implementations of AES are present in most general purpose cryptographic libraries and in a wide variety of programming languages, such as OpenSSL and OpenPGP (in C) or BouncyCastle (in Java). More specific, target-optimized implementations are found across the literature for a full spectrum of devices and architectures, ranging from ultra-constrained 4-bit CPUs [183] to multicore graphics processors (GPUs) [164].

In the rest of this section we give a brief review of state-of-the-art implementations targeted to embedded processors (8-bit and 32-bit) and to general purpose processors (32-bit and 64-bit). Optimizations often balance the common tradeoff speed vs. memory usage (RAM and/or ROM), and can be performed either at high-level or at low-level, i.e., by describing alternative reformulations of the AES building blocks or by exploiting microarchitectural features specific to the target CPU. The former techniques can typically be ported to a wide range of devices; whereas, the latter improvements are only valid for particular architectures.

© 2016 by Taylor & Francis Group, LLC

Downloaded By: 10.2.98.160 At: 00:35 27 Oct 2020; For: 9781482236897, chapter3, 10.1201/b19499-4

Speed reports vary significantly depending on the benchmarking method employed and on the choice of mode of operation. Direct comparison between proposals is thus often impossible, even if one considers only speed and not other parameters such as ROM or RAM requirements. In order to partially tackle this issue, the public eSTREAM benchmarking interface brings forward a common framework for comparison of software cryptographic implementation. For AES in particular, a detailed overview (in terms of speed) of existing proposals when considering counter mode of operation (AES-CTR) can be found in [37].

3.8.1 8-Bit Processors

This type of embedded processor is often found as the central element of smart cards, and is thus one of the most common targets for cryptographic implementations. Smart cards are generally constrained in program memory (flash or ROM), RAM, clock speed, or even arithmetic capabilities, and implementers have to balance and adapt these parameters according to their design goals. As the wordsize of 8-bit processors perfectly matches the basic unit for processing in the AES algorithm, namely a byte, a one-to-one mapping from the AES specification to its implementation is possible.

The non-linear transformation of the AES or `SubBytes` step is typically implemented as a lookup table (S-box), thus avoiding complex operations at the cost of storing a 256-byte array in memory. Aligning this table in a 256-byte memory boundary, e.g., starting at memory address 0x100, 0x200, etc., often results in cycle savings when updating the address pointer. The `AddRoundKey` step can be straightforwardly implemented as most 8-bit processors support additions in $GF(2^8)$ via the XOR (exclusive-or) instruction, while the `ShiftRows` step requires only to transpose the AES state bytes.

The matrix multiplication in the `MixColumns` step is often rewritten in a small series of instructions [94] that employ only field additions and field multiplications by 02:

$$
\begin{array}{lll}
t = a[0] \oplus a[1] \oplus a[2] \oplus a[3]; & \text{a is a 4-byte column} & \\
u = a[0]; & & \\
v = a[0] \oplus a[1]; & v = xtime(v); & a[0] = a[0] \oplus v \oplus t; \\
v = a[1] \oplus a[2]; & v = xtime(v); & a[1] = a[1] \oplus v \oplus t; \\
v = a[2] \oplus a[3]; & v = xtime(v); & a[2] = a[2] \oplus v \oplus t; \\
v = a[3] \oplus u; & v = xtime(v); & a[3] = a[3] \oplus v \oplus t;
\end{array}
$$

Multiplications by 02 in $GF(2^8)$, denoted as *xtime*, can be carried out in two different ways. One option is to execute a left-shift operation of the input value v and, if carry occurs, perform an additional XOR operation with the value $1B$, which corresponds to the AES irreducible polynomial. A second option consists in implementing *xtime* as a lookup table, thus avoiding conditional executions that might enable timing attacks.

© 2016 by Taylor & Francis Group, LLC

Downloaded By: 10.2.98.160 At: 00:35 27 Oct 2020; For: 9781482236897, chapter3, 10.1201/b19499-4

Regarding the decryption process, an efficient method to compute the more complex `InvMixColumns` step making only use of *xtime* can be also found in [94].

3.8.2 32-Bit Processors

A well-known technique to speed up AES implementations on 32-bit processors is the use of the so-called T-tables [94]. In a nutshell, this optimization allows execution of all AES round transformations with only four table lookups and four XOR operations per column. It requires 4 kB of storage space, which can be reduced to 1 kB at the cost of some extra rotations. The same technique can be efficiently applied for the decryption process, albeit using different lookup tables.

Further implementations on 32-bit processors include the work of Bertoni et al. [41]. They propose to restructure the AES algorithm in a different way than its standard formulation in order to allow a better exploitation in 32-bit processors. The proposed modifications result in considerable performance improvements in decryption on various 32-bit platforms such as ST22 smart cards, ARM7TDMI and ARM9TDMI processors, and Intel PentiumIII general purpose processors. Atasu et al. [22] exploit peculiarities of the 32-bit ARM instruction set architecture and propose a new implementation for the AES encryption linear mixing layers. Darnall and Kuhlman [95] explore various implementation tradeoffs on ARM7TDMI platforms focusing on three typical critical counts for embedded devices: execution time, ROM, and RAM.

3.8.3 64-Bit Processors

One of the first implementations of AES on the Intel x64 architecture was due to Matsui and Nakajima [279]. They reported a constant-time implementation with a throughput of 9.2 cycles/byte on an Intel Core 2, conditioned to the use of 2048-byte input data sizes previously transposed to a bitsliced format. Around 1 cycle/byte is required for bitsliced format conversion. A downside of this proposal is that smaller input sizes need to be padded to a 2048-byte boundary, thus creating significant overheads. The semi-bitsliced implementation due to Könighofer [232] reduces the minimum input data size to 64 bytes, but the throughput drops to 19.81 cycles/byte on an Athlon 64. Bernstein and Schwabe [38] report 10.57 cycles/byte on an Intel Core 2 and 10.43 cycles/byte on an Athlon 64.

A more recent bitsliced AES implementation due to Käsper and Schwabe [199] achieves 7.59 cycles/byte on an Intel Core 2 and 6.92 cycles/byte on Intel Core i7. The minimum input data size is 128-bytes which is stored into eight 128-bit XMM registers. This implementation takes advantage of Intel's microarchitectural SSSE3 instruction set extensions to improve results by a factor 30% compared to other implementations.

© 2016 by Taylor & Francis Group, LLC

Downloaded By: 10.2.98.160 At: 00:35 27 Oct 2020; For: 9781482236897, chapter3, 10.1201/b19499-4

3.9 Hardware Implementations of AES

The computational complexity of mapping modular Galois-field arithmetic onto general-purpose processors create power and performance bottlenecks, especially when high-throughput real-time operation is required. Special-purpose hardware AES accelerators address the need for energy-efficiency real-time AES encryption/decryption. Special-purpose hardware implementations of the AES algorithm started to appear in the literature at around the same time that the AES algorithm was being chosen as the standard by NIST in 2001 [441]. The efficiency of hardware implementations of AES is determined largely by how well the S-box inverse and MixColumn steps are implemented. The hardware implementation is routing-limited so an additional parameter to consider in the hardware implementation is the wire routing strategy. For example, AES instructions on Intel's Haswell platform requires 0.63 and 4.44 cycles/byte for CTR and CBC-encryption respectively, which when converted to Gb/Joule is many orders of magnitude more energy-efficient than generic assembly or C-compiled code [440]. The most challenging components of implementing AES using hardware are computing the multiplicative inverse in the SBox and routing the computational elements of the SBox and Mixed Columns. In this section we review three key AES hardware implementations and discuss future directions.

3.9.1 AES SBox Optimization for Hardware

The first efficient hardware implementation of AES was reported in [455] which describes optimizations to the AES Sbox inverse operation. Inversion in the finite field $GF(2^8)$ is calculated using combinational logic in the finite field $GF(2^4)$ which reduces the complexity of the inverse computation significantly. This implementation is contrasted with previous implementations that rely on look-up tables. The work presented a hardware implementation of the SBox which results in an area of $0.108mm^2$ in a fairly old CMOS 0.6μm process.

The AES hardware implementation area and power is largely determined by the implementation of the MixColumn operation and the SBox operation. The remaining operations are trivially implemented using XOR gates, shifts, and clever routing techniques. This work shows that the implementation significantly reduces area over a ROM implementation of the SBox inversion step. The basic idea that is employed in this work and later on leveraged in subsequent publications relies in representing the finite field $GF(2^8)$ AES generator polynomial in a composite field $GF((2^4)^2)$. The authors show that this can be done because the finite field $GF(2^8)$ is isomorphic to the finite field $GF((2^4)^2)$, where for each element in $GF(2^8)$ there exists exactly one element in $GF((2^4)^2)$. The representation and mapping are explained mathematically

© 2016 by Taylor & Francis Group, LLC

Downloaded By: 10.2.98.160 At: 00:35 27 Oct 2020; For: 9781482236897, chapter3, 10.1201/b19499-4

in the above referred paper and will not be repeated here. The implementation requires simple gates such as inverters, XORs, and multiplexers.

3.9.2 AES for Microprocessors

The second noteworthy implementation that we review was reported in 2010 [276] where the modular inversion is also carried out in the composite $GF(2^4)^2$ field which enables an efficient circuit implementation at the cost of mapping/inverse-mapping overhead. However, in this implementation the mapping overhead is amortized over all the AES rounds by moving the transformation outside of the iteration loop. This is accomplished by computing the entire AES round in the composite field $GF(2^4)^2$ in contrast to most conventional implementations that compute only the SBox in the composite field as presented in the previous section. The implementation of the SBox and the modular inverse is computed by combinational logic as in [455] with further optimizations in the logic resulting in a smaller gate count. The other features of this implementation include a reconfigurable datapath for encrypt/decrypt, optimized ground and composite-field polynomials, integration of bypass multiplexers into affine transforms, difference-matrix computation for the InvMixColumn step, and a folded ShiftRow datapath organization. Careful consideration has been given to wire routing to optimize layout area.

The 53Gbps AES encrypt/decrypt hardware accelerator is manufactured in 45nm CMOS process resulting in 67% reduction in interconnect length and 20% reduction in total area. The accelerator is reconfigurable to support different modes of AES ECB: AES-128, AES-192, and AES-256. The implementation consumes 125mW total power measured at 1.1V. This state-of-the-art implementation at the time was accomplished by amortizing the mapping the entire AES round to the composite field $GF(2^4)^2$ and amortizing this mapping over the multiple iterations required. The authors implemented a fused MixColumns/InverseMixColumns circuit to save area and folded the SBox datapath to reduce wire routing overhead. The reader is referred to the authors' detailed paper in [276] to fully understand the mechanisms and design techniques required to achieve high-throughput required for high-performance microprocessors.

3.9.3 AES for Ultra-Low Power Mobile Platforms

In contrast to the above implementation targeted towards high-performance computing, there is an increasing requirement for encrypted data storage and communication across low-power devices such as RFID tags, Internet-of-Things (IOT) compute devices, and even mobile phones and wearables. AES has become the defacto block cipher for content protection, memory encryption, and network security. However, the large area cost and high energy consumption for parallel, multiple round implementation is prohibitive

© 2016 by Taylor & Francis Group, LLC

Downloaded By: 10.2.98.160 At: 00:35 27 Oct 2020; For: 9781482236897, chapter3, 10.1201/b19499-4

for small embedded devices. The first application of hardware accelerated AES was in fact for RFID applications where encryption was necessary to secure and protect data. In [121] the authors report an AES hardware implementation which encrypts 128-bit block of data within 1000 clock cycles and consumes below 9μA on 0.35μm CMOS process. Ingrid Verbauwhede is famous for implementing low-cost cryptographic processing hardware solutions starting from AES for RFIDs, with the first low-power implementation presented in 2002 [237]. The most recent attempt at implementing an ultra-low power AES hardware accelerator was presented in 2014 [275].

In this most recent implementation of AES targeted towards ultra-low power encryption/decryption, the authors present 2090-gate AES-128 encrypt/decrypt engine fabricated in 22nm CMOS consuming 13mW at 1.1GHz operational frequency at 0.9V. Optimum energy efficiency is achieved at a supply voltage of 430mV which results in energy-efficiency of 289Gbps/W. The key to implementing a low-power version of AES is to serialize the computation of the SBox. That is, only implement one SBox and reuse it over time. The authors in [275] have also mathematically optimized the implementation by comprehensive exploration of the $GF(2^4)^2)$ polynomial space to select non-symmetric ground and extension-field polynomials for encrypt and decrypt that minimize layout area by up to 9% area reduction over previous work.

3.10 Future Directions

There is continuing work to develop extreme low-power implementations of the now defacto AES standard used for encryption of media content, memory and disk storage, and network communication. Body Area Networks (BAN) are now emerging as the next phase in personal data communication over wireless networks where it is essential to protect privacy and secure data communication. However, in this space, extremely low energy solutions are necessary. Whether it is AES that can be implemented in a "nano" regime or whether it is other ultra-low cost block ciphers, there will be continuing efforts to develop new algorithms and implementations to meet the growing demand for secure data storage and transmission.

© 2016 by Taylor & Francis Group, LLC

Downloaded By: 10.2.98.160 At: 00:35 27 Oct 2020; For: 9781482236897, chapter3, 10.1201/b19499-4