

Chapter 2

Modern Cryptography

The art of cryptographic algorithms is an ever evolving field. Initiating from prehistoric times, the main objective of cryptographic algorithms have been to protect and allow usage of information in a legal manner. Encryption is a process of converting a plaintext message into a piece of random-looking text, often called ciphertext. The ciphertext, ideally, should contain or transfer no information to the curious third party, often referred to as the adversary. In order to allow the intended receiver to obtain back the message from the ciphertext, every encryption algorithm is reversible. Thus, the inverse operation of obtaining back the plaintext from the ciphertext, is called decryption. According to the wisdom in cryptography, the algorithms for encryption and decryption are always published and known to even the adversary. Then what does the security rely on? The mappings (from plaintext to ciphertext and vice versa) depends on an information called the key, which is hidden from the attacker. While the goal of cryptography is to design and construct such ciphering algorithms, the objective of cryptanalysis is to develop techniques to obtain the key more efficiently than making a random guess on the key. There are different classes of cryptographic algorithms, depending on their objectives. While secrecy is the obvious requirement, there are other important goals too; for example, algorithms to guarantee the integrity of information, or methods to ensure that one cannot deny a commitment to a transaction (often called the property of nonrepudiation). Then there are algorithms which ensures that authenticity is maintained in a communication, and legal parties can trust with whom they are communicating. In this book we mainly target cryptographic algorithms, with respect to secrecy of data. But often these constructions can be used for achieving the other requirements, namely authentication and nonrepudiation. Hash functions, which are used for integrity checks are not in the scope of this book.

2.1 Types of Encryption Algorithms

For ciphers, there is an encryption key (K_a) and a decryption key (K_b), which are equal for certain classes of algorithms (called symmetric-key ciphers) and different for the other class (called asymmetric-key ciphers). The scenario of a cryptographic

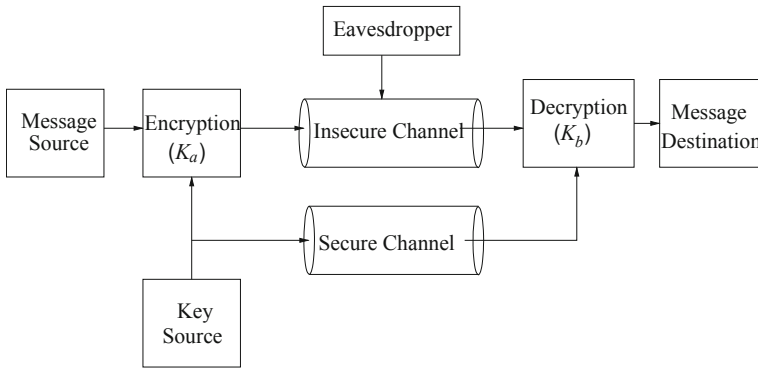


Fig. 2.1 Secret key cryptosystem model

communication is illustrated in Fig. 2.1. The encryptor uses a key K_a and the decryptor a key K_b , where depending on the equality of K_a and K_b there are two important classes of cryptographic algorithms. More precisely, the two classes of ciphers are:

- Private-key (or symmetric) ciphers: These ciphers have the same key shared between the sender and the receiver. Thus, referring to Fig. 2.1 $K_a = K_b$.
- Public-key (or asymmetric) ciphers: In these ciphers we have $K_a \neq K_b$. The encryption key and the decryption keys are different.

As in symmetric-key or private-key algorithms both the encryptor and decryptor use the same key, it must somehow be securely exchanged before secret key communication can begin. The key exchange is a major bottleneck and for n -parties in a network, the number of key exchanges required can grow quite fast (nC_2 ways). However, these algorithms are often fast and are used for bulk data encryption. Two important subclasses of symmetric-key algorithms are block ciphers and stream ciphers. Block ciphers, as the name suggest operates on fixed blocks or chunks of data, while stream ciphers operate on bits or few bits of data (thus the encryption takes place like a stream!). The Advanced Encryption Standard (AES) is a very popular block cipher, while Trivium is a popular stream cipher.

Public-key algorithms, on the other hand, provide a nice solution to the key-exchange problem. In such algorithms, as we discussed the encryption and decryption keys are different. The algorithms have a key pair, consisting of (i) Public key, which can be freely distributed and is used to encrypt messages. In Fig. 2.1, this is denoted by the key K_a , and (ii) Private key, which must be kept secret and is used to decrypt messages. The decryption key is denoted by K_b in the Fig. 2.1.

In the public key or asymmetric ciphers, the two parties—often called Alice and Bob—are communicating with each other and have their own key pair. They distribute their public keys freely. Mallory (or the adversary) has the knowledge of not only the encryption function, the decryption function, and the ciphertext, but also has the capability to encrypt the messages using Bob's public key. However, she is unaware of the secret decryption key, which is the private key of the algorithm. The security

of these classes of algorithms rely on the assumption that it is computationally hard or complex to obtain the private key from the public information. Doing so would imply that the adversary solves a mathematical problem which is widely believed to be difficult. It may be noted that we do not have any proofs for their hardness; however, we are unaware of any efficient techniques to solve them. The elegance of constructing these ciphers lies in the fact that the public keys and private keys still have to be related in the sense, that they perform the invertible operations to obtain the message back. This is achieved through a class of magical functions, which are called *one-way* functions. These functions are easy to compute in one direction, while computing the inverse from the output is believed to be a difficult problem. RSA is a famous public-key algorithm for this class of ciphers.

Example 2.1 This cipher is called the famous RSA algorithm (Rivest Shamir Adleman). Let $n = pq$, where p and q are properly chosen and large prime numbers. Here the proper choice of p and q are to ensure that factorization of n is mathematically complex. The plaintexts and ciphertexts are $P = C = \mathbb{Z}_n$, the keys are $K_a = \{n, a\}$ and $K_b = \{b, p, q\}$, such that $ab \equiv 1 \pmod{\phi(n)}$. The encryption and decryption functions are defined as, $\forall x \in P, e_{K_a}(x) = y = x^a \pmod n$ and $d_{K_b}(y) = y^b \pmod n$.

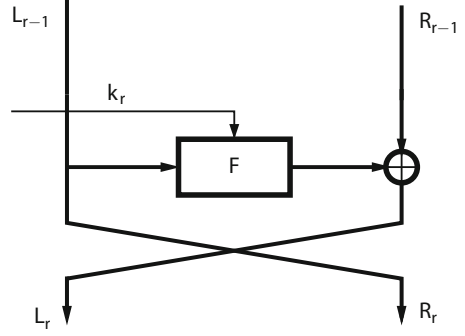
Both symmetric-key ciphers and asymmetric-key ciphers are widely studied. We provide a quick overview on some facts, which we use in the book.

2.2 Block Ciphers: An Important Family of Symmetric-Key Ciphers

The book develops a general theory for time-driven cache attacks on block ciphers. The theory can be applied to any cipher with an iterative structure that is implemented with look-up tables. To test the theory we selected few ciphers, like AES, CAMELLIA, and CLEFIA. AES was chosen because it is the world wide standard. It has a substitution permutation network (SPN) structure and the implementation generally used has 5 large tables of 1024 bytes. The first four tables are invoked 36 times per 128-bit encryption, while the last table is invoked 16 times. Many of the latest CPUs support dedicated instructions for AES [1], on which cache attacks fail. Besides SPN, most cipher designs follow the Feistel structure (Fig. 2.2). The figure shows the r th round of a Feistel block cipher. The block is divided into two parts, L_r (Left) and R_r (Right), which are recursively computed as $L_r = R_{r-1} \oplus \mathbf{F}(L_{r-1}, k_r)$, and $R_r = L_{r-1}$. The transformation \mathbf{F} is composed of several nonlinear transformations or S-Boxes, and combine the round key k_r with a portion of the block. The reversibility of the round does not depend on whether the nonlinear layer \mathbf{F} is invertible or not. The rounds are repeated for certain number of iterations to ensure sufficient security margin against known attacks.

We chose CAMELLIA as a representative of these ciphers. The third cipher we chose is CLEFIA, which has a generalized Feistel structure [2] and a round function which is conceptually similar to that of AES. Further, unlike the AES implementation considered, CLEFIA and CAMELLIA implementations used small tables of 256 bytes. This section provides a brief description of each cipher algorithm.

Fig. 2.2 Feistel structure of a block cipher



2.2.1 AES

In 2001, the National Institute of Standards and Technology (NIST) recommended the use of Rijndael as the AES [3]. AES is a symmetric-key block cipher and can use key sizes of either 128, 192, or 256 bits to encrypt and decrypt 128-bit blocks. We summarize the AES-128 standard, which uses a key size of 128 bits. The input to AES-128 is arranged in a 4×4 matrix of bytes called *state*. The state undergoes a series of transformations in ten rounds during the encryption process.

Algorithm 2.1 presents the AES-128 algorithm. The first operation on the input is the **AddRoundKeys**, which serves to provide the initial randomness by mixing the input key. The state is then subjected to nine rounds to further increase the diffusion and confusion in the cipher [4]. Each round comprises four operations on the state: **SubBytes**, **ShiftRows**, **MixColumns**, and **AddRoundKeys**. The state is then subjected to a final round, which has all operations except the **MixColumns** operation.

The four AES operations are defined as follows:

Algorithm 2.1: AES-128

Input: 128 bit plaintext input block \mathbf{x} and 128 bit round keys $\mathbf{k}^{(0)}, \mathbf{k}^{(2)}, \dots, \mathbf{k}^{(10)}$ generated from a secret key using the AES-128 key generation algorithm

Output: 128 bit ciphertext

```

1 begin
2    $r \leftarrow 1$ 
3    $\mathbf{s}^{(0)} \leftarrow \text{AddRoundKeys}(\mathbf{x}, \mathbf{k}^{(0)})$ 
4   while  $r \leq 9$  do
5      $\mathbf{s}^{(r)} \leftarrow \text{SubBytes}(\mathbf{s}^{(r-1)})$ 
6      $\mathbf{s}^{(r)} \leftarrow \text{ShiftRows}(\mathbf{s}^{(r)})$ 
7      $\mathbf{s}^{(r)} \leftarrow \text{MixColumns}(\mathbf{s}^{(r)})$ 
8      $\mathbf{s}^{(r)} \leftarrow \text{AddRoundKeys}(\mathbf{s}^{(r)}, \mathbf{k}^{(r)})$ 
9   end
10   $\mathbf{s}^{(10)} \leftarrow \text{SubBytes}(\mathbf{s}^{(9)})$ 
11   $\mathbf{s}^{(10)} \leftarrow \text{ShiftRows}(\mathbf{s}^{(10)})$ 
12   $\mathbf{y} \leftarrow \text{AddRoundKeys}(\mathbf{s}^{(10)}, \mathbf{k}^{(10)})$ 
13  return  $\mathbf{y}$ 
14 end

```

- **AddRoundKeys** : Each element in the state is subjected to a bitwise ex-or with a 128-bit round key. The round key is generated from the secret key by a key expansion algorithm as in Algorithm 2.2 [4].
- **SubBytes** : Each element in the state is replaced by an affine transformation of its inverse in the field $GF(2^8)$. For a byte s_i in the state, this operation is denoted by $S(s_i)$.
- **ShiftRows** : Provides a cyclic shift of the i th row in the state by i bytes toward the left (where $0 \leq i \leq 3$). That is, each byte in the i th row is cyclically shifted to the left by i bytes.
- **MixColumns** : Provides a column-wise linear transformation of the state matrix. Each column of the state matrix is considered as a polynomial of degree 3 with coefficients in $GF(2^8)$ and multiplied by the polynomial $\{03\}\alpha^3 + \{01\}\alpha^2 + \{01\}\alpha + \{02\} \pmod{\alpha^4 + 1}$. The combination of **ShiftRows** and **MixColumns** provide the necessary diffusion for the cipher.

Key Expansion Algorithm [3] takes the secret key as input and generates round keys for 11 **AddRoundKeys** operations performed in AES. Key expansion as in Algorithm 2.2 uses two operations **ROTWORD** and **SUBWORD** which performs cyclic shift and substitution of four bytes (B_0, B_1, B_2, B_3) as

$$ROTWORD(B_0, B_1, B_2, B_3) = (B_1, B_2, B_3, B_0) \quad (2.1)$$

and

$$SUBWORD(B_0, B_1, B_2, B_3) = (SubBytes(B_0), SubBytes(B_1), SubBytes(B_2), SubBytes(B_3)) \quad (2.2)$$

Algorithm 2.2: Key Expansion of AES-128

Input: 128 bit secret key

Output: 11 round keys each of 4 words as $w[0], \dots, w[43]$

```

1  begin
2     $RCon[1] \leftarrow 01000000$ 
3     $RCon[2] \leftarrow 02000000$ 
4     $RCon[3] \leftarrow 04000000$ 
5     $RCon[4] \leftarrow 08000000$ 
6     $RCon[5] \leftarrow 10000000$ 
7     $RCon[6] \leftarrow 20000000$ 
8     $RCon[7] \leftarrow 40000000$ 
9     $RCon[8] \leftarrow 80000000$ 
10    $RCon[9] \leftarrow 1B000000$ 
11    $RCon[10] \leftarrow 36000000$ 
12   for  $i \leq 0$  to 3 do
13      $w[i] \leftarrow (k[4i], k[4i+1], k[4i+2], k[4i+3])$ 
14   end
15   for  $i \leq 4$  to 43 do
16      $temp \leftarrow w[i-1]$ 
17     if  $i \equiv 0 \pmod{4}$  then
18        $temp = SUBWORD(ROTWORD(temp)) \oplus RCon[i/4]$ 
19     end
20      $w[i] \leftarrow w[i-4] \oplus temp$ 
21   end
22   return  $(w[0], \dots, w[43])$ 
23 end
```

In addition to this, AES involves a round constant term that is defined as $RCon[1], \dots, RCon[10]$ as constants in hexadecimal before the key expansion Algorithm 2.2.

Starting from the 4×4 byte state, Fig. 2.3 shows the transformation it undergoes in a round (for $1 \leq r \leq 9$).

2.2.1.1 Software Implementations of AES

Of all operations, the **SubBytes** is the most difficult to implement. On 8-bit micro-controllers, a 256-byte look-up table is ideal to perform this operation. The table provides the necessary flexibility in terms of content, small footprint, and speed. For 32-bit platforms, more efficient implementations can be built using larger tables. We give a brief description of this method, which is known as *T-table* implementations. *T-table* implementations were first proposed in [5] and has been adopted by several crypto-libraries such as OpenSSL¹.

Consider four look-up tables defined as follows:

$$\begin{aligned} \mathcal{T}_0[z] &= \begin{bmatrix} 02 \bullet S(z) \\ S(z) \\ S(z) \\ 03 \bullet S(z) \end{bmatrix}; \mathcal{T}_1[z] = \begin{bmatrix} 03 \bullet S(z) \\ 02 \bullet S(z) \\ S(z) \\ S(z) \end{bmatrix}; \mathcal{T}_2[z] = \begin{bmatrix} S(z) \\ 03 \bullet S(z) \\ 02 \bullet S(z) \\ S(z) \end{bmatrix}; \\ \mathcal{T}_3[z] &= \begin{bmatrix} S(z) \\ S(z) \\ 03 \bullet S(z) \\ 02 \bullet S(z) \end{bmatrix} \end{aligned} \quad (2.3)$$

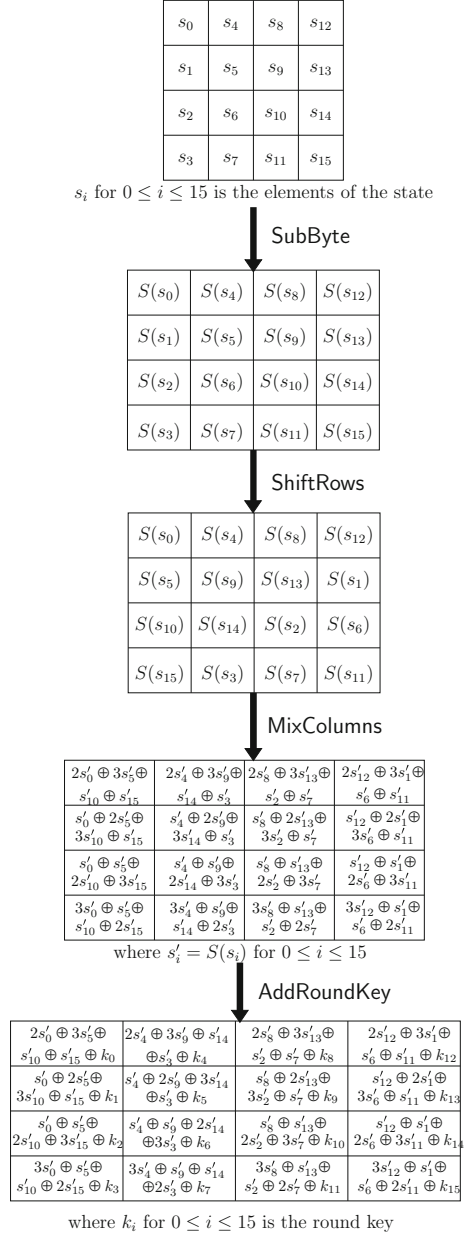
Each table is of 1024 bytes mapping a byte z of the state to a 32-bit value. Using these tables, the first nine AES rounds can be expressed as follows

$$\begin{aligned} \mathbf{s}^{(r+1)} &= \mathcal{T}_0[s_0^{(r)}] \oplus \mathcal{T}_1[s_5^{(r)}] \oplus \mathcal{T}_2[s_{10}^{(r)}] \oplus \mathcal{T}_3[s_{15}^{(r)}] \oplus [k_0^{(r)} \ k_1^{(r)} \ k_2^{(r)} \ k_3^{(r)}]^T \parallel \\ &\quad \mathcal{T}_0[s_4^{(r)}] \oplus \mathcal{T}_1[s_9^{(r)}] \oplus \mathcal{T}_2[s_{14}^{(r)}] \oplus \mathcal{T}_3[s_3^{(r)}] \oplus [k_4^{(r)} \ k_5^{(r)} \ k_6^{(r)} \ k_7^{(r)}]^T \parallel \quad (2.4) \\ &\quad \mathcal{T}_0[s_8^{(r)}] \oplus \mathcal{T}_1[s_{13}^{(r)}] \oplus \mathcal{T}_2[s_2^{(r)}] \oplus \mathcal{T}_3[s_7^{(r)}] \oplus [k_8^{(r)} \ k_9^{(r)} \ k_{10}^{(r)} \ k_{11}^{(r)}]^T \parallel \\ &\quad \mathcal{T}_0[s_{12}^{(r)}] \oplus \mathcal{T}_1[s_1^{(r)}] \oplus \mathcal{T}_2[s_6^{(r)}] \oplus \mathcal{T}_3[s_{11}^{(r)}] \oplus [k_{12}^{(r)} \ k_{13}^{(r)} \ k_{14}^{(r)} \ k_{15}^{(r)}]^T \end{aligned}$$

A byte of the state in the current round ($\mathbf{s}^{(r)}$) is denoted $s_i^{(r)}$ and the next round state is denoted $\mathbf{s}^{(r+1)}$, where $0 \leq r \leq 9$ and $0 \leq i \leq 15$. The final round cannot use these tables due to the absence of the **MixColumns** operation.

¹ <http://www.openssl.org>.

Fig. 2.3 Transformations of the state in a round of AES ($1 \leq r \leq 9$)



2.2.2 CLEFIA

CLEFIA is a 128-bit block cipher developed by Sony [6] and is currently incorporated in ISO/IEC 29192-2 as a light-weight block cipher standard. The specification [6, 7] defines three key lengths of 128, 192, and 256 bits. This book considers 128-bit keys though the results are valid for the other key sizes also. The structure of CLEFIA is shown in Fig. 2.4. The input has 16 bytes, P_0 to P_{15} , grouped into four byte words. There are 18 rounds, and in each round, the first and third words are fed into nonlinear functions $F0$ and $F1$ respectively. The output of $F0$ and $F1$ are ex-ored with the second and fourth words. Additionally, the second and fourth words are also whitened at the beginning and end of the encryption. The F functions take four input bytes and four round keys. The nonlinearity in the F functions are due to two 256 element s-boxes $S0$ and $S1$. Matrices $M0$ and $M1$ diffuse the outputs of the s-boxes. They are defined as follows:

$$M0 = \begin{pmatrix} 1 & 2 & 4 & 6 \\ 2 & 1 & 6 & 4 \\ 4 & 6 & 1 & 2 \\ 6 & 4 & 2 & 1 \end{pmatrix} \quad M1 = \begin{pmatrix} 1 & 8 & 2 & A \\ 8 & 1 & A & 2 \\ 2 & A & 1 & 8 \\ A & 2 & 8 & 1 \end{pmatrix} \quad (2.5)$$

The design of the s-boxes $S0$ and $S1$ differs. $S0$ is composed of four s-boxes $SS0$, $SS1$, $SS2$, and $SS3$; each of 16 bytes. The output of $S0$ is given by :

$$\begin{aligned} \beta_l &= SS2[SS0[\alpha_l] \oplus 2 \cdot SS1[\alpha_h]] \\ \beta_h &= SS3[SS1[\alpha_h] \oplus 2 \cdot SS0[\alpha_l]] \end{aligned} \quad (2.6)$$

where $\beta = (\beta_h | \beta_l)$, $\alpha = (\alpha_h | \alpha_l)$, and $\beta = S0[\alpha]$. The output of $S1$ for the input byte α is given by $g((f(\alpha))^{-1})$, where g and f are affine transforms and the inverse is found in the field $GF(2^8)$.

The CLEFIA encryption algorithm has four whitening keys WK_0, WK_1, WK_2 , and WK_3 ; and 36 round keys RK_0, \dots, RK_{35} . Key expansion is a two-step process. First, a 128-bit intermediate key L is generated from the secret key K using a GFN function [7]. From this, the round keys and whitening keys are generated as shown below:

```

Step 1:  $WK_0 | WK_1 | WK_2 | WK_3 \leftarrow K$ 
Step 2: For  $i \leftarrow 0$  to 8
     $T \leftarrow L \oplus (CON_{24+4i} | CON_{24+4i+1} | CON_{24+4i+2} | CON_{24+4i+3})$ 
     $L \leftarrow \Sigma(L)$ 
    if  $i$  is odd:  $T \leftarrow T \oplus K$ 
     $RK_{4i} | RK_{4i+1} | RK_{4i+2} | RK_{4i+3} \leftarrow T$ 
```

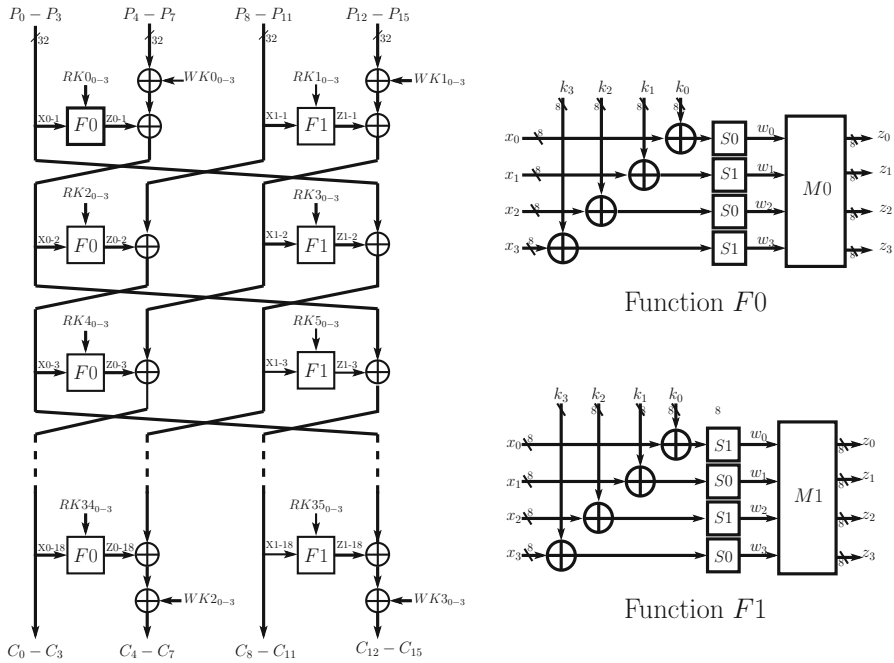
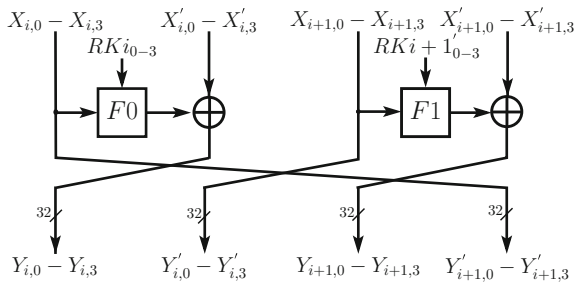



Fig. 2.4 CLEFIA block diagram

Fig. 2.5 A round of CLEFIA



The function Σ , known as the *double swap* function, rearranges the bits of L .

$$\Sigma(L) \leftarrow L_{(7 \dots 63)} | L_{(121 \dots 127)} | L_{(0 \dots 6)} | L_{(64 \dots 120)} \quad (2.7)$$

In the later part, thirty-six 32-bit constant values CON_i ($24 \leq i < 60$) are used.

Just like AES, CLEFIA can be implemented with T -tables. The T -table implementation for CLEFIA is in the Sect. 2.2.2.1.

2.2.2.1 T -Table Implementation of CLEFIA

A round of CLEFIA has 128-bit input and produces 128-bit output. Each input and output is grouped as four words as shown in Fig. 2.5. The function $F0$ is defined as

follows,

$$F0 : \begin{bmatrix} Z_{i,0} \\ Z_{i,1} \\ Z_{i,2} \\ Z_{i,3} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 4 & 6 \\ 2 & 1 & 6 & 4 \\ 4 & 6 & 1 & 2 \\ 6 & 4 & 2 & 1 \end{bmatrix} \times \begin{bmatrix} S0(X_{i,0} \oplus RK_{i,0}) \\ S1(X_{i,1} \oplus RK_{i,1}) \\ S0(X_{i,2} \oplus RK_{i,2}) \\ S1(X_{i,3} \oplus RK_{i,3}) \end{bmatrix}$$

where $S0$ and $S1$ are CLEFIA's 8×8 s-boxes. $X_{i,k}$ and $Z_{i,k}$ represent a byte of the input and output word respectively ($0 \leq k \leq 3$), while $RK_{i,k}$ is a byte of the round key. The matrix multiplication is in a finite field.

Consider four look-up tables as follows which map an input byte (x) to a 32-bit word.

$$\begin{aligned} T_0[x] &\leftarrow (6 \cdot S0(x)|4 \cdot S0(x)|2 \cdot S0(x)|1 \cdot S0(x)) \\ T_1[x] &\leftarrow (4 \cdot S1(x)|6 \cdot S1(x)|1 \cdot S1(x)|2 \cdot S1(x)) \\ T_2[x] &\leftarrow (2 \cdot S0(x)|1 \cdot S0(x)|6 \cdot S0(x)|4 \cdot S0(x)) \\ T_3[x] &\leftarrow (1 \cdot S1(x)|2 \cdot S1(x)|4 \cdot S1(x)|6 \cdot S1(x)). \end{aligned}$$

Then,

$$\begin{aligned} (Z_{i,3}|Z_{i,2}|Z_{i,1}|Z_{i,0}) &= T_0[X_{i,0} \oplus RK_{i,0}] \oplus T_1[X_{i,1} \oplus RK_{i,1}] \\ &\quad \oplus T_2[X_{i,2} \oplus RK_{i,2}] \oplus T_3[X_{i,3} \oplus RK_{i,3}] \end{aligned}$$

The function $F1$ is defined as follows:

$$F1 : \begin{bmatrix} Z_{i+1,0} \\ Z_{i+1,1} \\ Z_{i+1,2} \\ Z_{i+1,3} \end{bmatrix} = \begin{bmatrix} 1 & 8 & 2 & A \\ 8 & 1 & A & 2 \\ 2 & A & 1 & 8 \\ A & 2 & 8 & 1 \end{bmatrix} \times \begin{bmatrix} S1(X_{i+1,0} \oplus RK_{i+1,0}) \\ S0(X_{i+1,1} \oplus RK_{i+1,1}) \\ S1(X_{i+1,2} \oplus RK_{i+1,2}) \\ S0(X_{i+1,3} \oplus RK_{i+1,3}) \end{bmatrix}$$

The function $F1$ can be implemented using four tables as shown below. Each table maps the 8-bit input x to a 32-bit output.

$$\begin{aligned} T_4[x] &\leftarrow (A \cdot S1(x)|2 \cdot S1(x)|8 \cdot S1(x)|1 \cdot S1(x)) \\ T_5[x] &\leftarrow (2 \cdot S0(x)|A \cdot S0(x)|1 \cdot S0(x)|8 \cdot S0(x)) \\ T_6[x] &\leftarrow (8 \cdot S1(x)|1 \cdot S1(x)|A \cdot S1(x)|2 \cdot S1(x)) \\ T_7[x] &\leftarrow (1 \cdot S0(x)|8 \cdot S0(x)|2 \cdot S0(x)|A \cdot S0(x)) \end{aligned}$$

Then,

$$\begin{aligned}
 (Z_{i+1,3}|Z_{i+1,2}|Z_{i+1,1}|Z_{i+1,0}) = & T_4[X_{i+1,0} \oplus RK_{i+1,0}] \\
 & \oplus T_5[X_{i+1,1} \oplus RK_{i+1,1}] \\
 & \oplus T_6[X_{i+1,2} \oplus RK_{i+1,2}] \\
 & \oplus T_7[X_{i+1,3} \oplus RK_{i+1,3}].
 \end{aligned}$$

The output of the round is

$$\begin{aligned}
 (Y_{i,0} \cdots Y_{i,3}) & \leftarrow (Z_{i,0} \cdots Z_{i,3}) \oplus (X'_{i,0} \cdots X'_{i,3}) \\
 (Y'_{i,0} \cdots Y'_{i,3}) & \leftarrow (X_{i+1,0} \cdots X_{i+1,3}) \\
 (Y_{i+1,0} \cdots Y_{i+1,3}) & \leftarrow (Z_{i+1,0} \cdots Z_{i+1,3}) \oplus (X'_{i+1,0} \cdots X'_{i+1,3}) \\
 (Y'_{i+1,0} \cdots Y'_{i+1,3}) & \leftarrow (X_{i,0} \cdots X_{i,3}).
 \end{aligned}$$

In a similar manner, all rounds of CLEFIA can be implemented. Thus reducing the implementation to a series of memory accesses intertwined with ex-ors.

2.2.3 CAMELLIA

CAMELLIA is the 128-bit block cipher that was jointly developed by Mitsubishi and NTT in 2000 [8]. Since the cipher has been made available under a royalty-free license, it has been certified for use by the European Union and Japan. It has also become part of the OpenSSL project, and incorporated in Mozilla's Network Security Services (NSS modules). Support for CAMELLIA has been added to several security libraries as well as Mozilla's popular web browser, Firefox.

The 128-bit block cipher CAMELLIA has a Feistel structure as shown in Fig. 2.6. The 16 bytes plaintext input is grouped in two words of 8 bytes each: $\mathbf{x} = (x_1 \| x_2 \| \cdots \| x_8)$ and $\mathbf{y} = (y_1 \| y_2 \| \cdots \| y_8)$. There are 18 rounds in all, broken up into groups of 6 each. After the 6th and the 12th rounds, there are two FL/FL^{-1} function layers. In each round, there is an F function, which is a combination of key addition, substitution (S), and permutation (P). The substitution is done by using four s-boxes, whereas, the P function is implemented by using a diffusion matrix. Figure 2.7 shows the permutation operation and the diffusion matrix. The diffusion matrix also has an inverse depicted in the figure.

Each round has an addition of a round key. The i th round uses the round key $\mathbf{k}^{(i)}$. Each of these round keys are of 64 bits. Additionally, whitening keys $kw1$ and $kw2$ are applied at the start of encryption, while $kw3$ and $kw4$ are applied at the end of encryption.

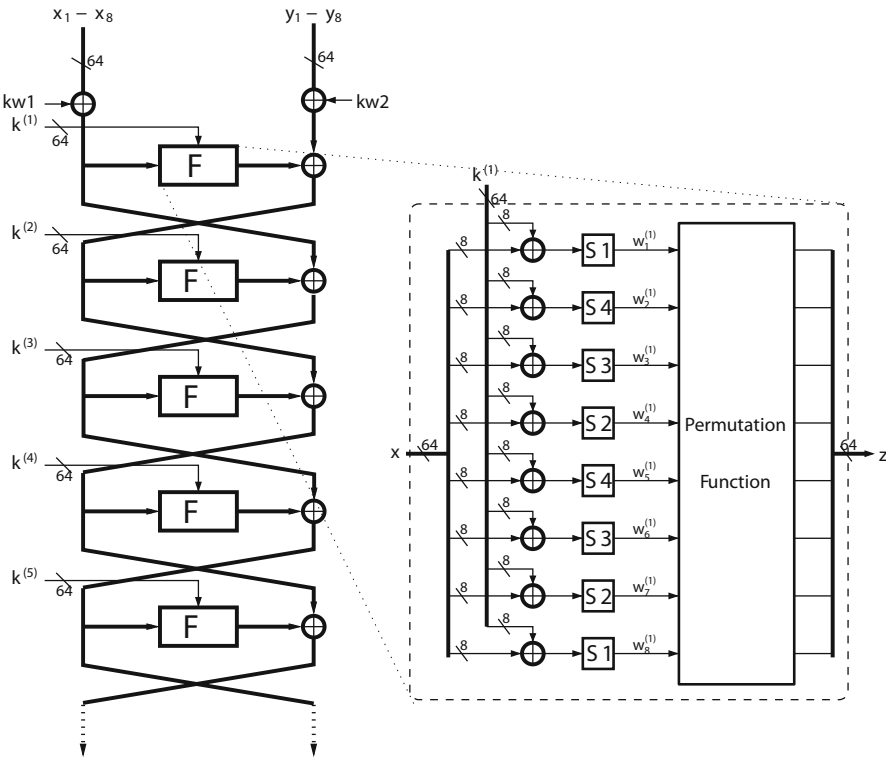


Fig. 2.6 Structure of CAMELLIA

Fig. 2.7 CAMELLIA's diffusion layer and its inverse

$$\begin{pmatrix} z_8 \\ z_7 \\ \vdots \\ z_1 \end{pmatrix} \leftarrow P \cdot \begin{pmatrix} w_8 \\ w_7 \\ \vdots \\ w_1 \end{pmatrix}$$

(a) Permutation Function

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

(b) Diffusion Matrix P

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

(c) Inverse Diffusion Matrix P^{-1}

2.3 Classical Cryptanalysis

A cryptanalytic attack is a procedure through which an attacker gains information about the secret decryption key. Attacks are classified according to the level of a priori knowledge available to the attacker.

A *ciphertext-only attack* is an attack where the cryptanalyst has access to ciphertexts generated using a given key but has no access to the corresponding plaintexts or the key. A *known-plaintext attack* is an attack where the cryptanalyst has access to both ciphertexts and the corresponding plaintexts but not the key.

A **chosen-plaintext attack** (CPA) is an attack where the cryptanalyst can choose plaintexts to be encrypted and has access to the resulting ciphertexts, again their purpose being to determine the key.

A **chosen-ciphertext attack** (CCA) is an attack in which the cryptanalyst can choose ciphertexts, apart from the challenge ciphertext and can obtain the corresponding plaintext. The attacker has access to the decryption device.

In case of CPA and CCA, adversaries can make a bounded number of queries to its encryption or decryption device. The encryption device is often called an oracle; meaning it is like a black-box without details like in an algorithm of how an input is transformed or used to obtain the output. Although this may seem a bit hypothetical, but there are enough real life instances where such encryption and decryption oracles can be obtained.

In the next section, we discuss one form of classical cryptanalysis of block ciphers, namely differential attacks, which is used in developing some of the cache attacks described in the book.

2.3.1 *Classical Cryptanalysis of Block Ciphers*

Block ciphers have been subjected to several forms of classical cryptanalysis; namely linear cryptanalysis, differential cryptanalysis, impossible differential attacks, related key attacks, boomerang attacks, square attacks are some popular cryptanalysis techniques. In this book, we study how microarchitectural features like cache memories and the accompanying hardware increases leakage when an encryption program runs on this platform. These attacks, often called as side-channel attacks (SCA) exploit additional leakage through timing information, and combine with classical methods, like differential attacks to provide efficient attack techniques. In this section, we provide a quick overview on the ideas of differential cryptanalysis, which will be useful for other attacks. Interested readers may refer [9] for ideas on other forms of cryptanalysis, which may (or may not) lead to more efficient cache attacks.

2.3.2 *The Idea of Differential in Block Ciphers*

Differential cryptanalysis is a chosen plain text attack and the attack is based on the information of the ex-or of two inputs, and the ex-or of corresponding outputs. Consider, a cipher expressed as $c = m \oplus k$, where m , k , and c are the plaintext, key, and the ciphertext blocks respectively, each of size b -bits. We know this is a secured cipher if the key is randomly and uniquely chosen for every encryption. The attacker has no information about the plaintext from the ciphertext. However, if the key is used twice for two encryptions there is a leakage of information. It is trivial to note, $c_0 \oplus c_1 = m_0 \oplus m_1$, thus showing that the *differential* (or, the difference which is computed using ex-ors between two blocks) does not depend on the key, and thus the key (although unknown) does not hide the plaintext from the attacker! The differential is often denoted by $\Delta(c) = c_0 \oplus c_1$. Thus, in brevity we can state

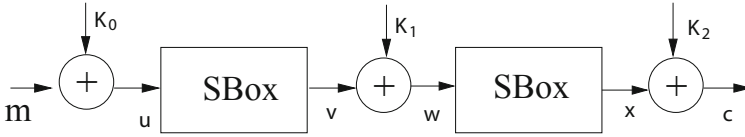


Fig. 2.8 Illustration of a differential attack on a toy cipher

Table 2.1 The S-Box description

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$s[x]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

$\Delta(c) = \Delta(m)$ for the above equality. It may be noted that if the key mixing was done by some other reversible process (like modular integer addition), the differential would be accordingly modified (by using modular integer subtraction). Let us see how we can use this fact to cryptanalyze block ciphers with S-Boxes.

In the Fig. 2.8, a two round encryption is depicted where the plaintext m is transformed by mixing (ex-oring) with the keys, k_0 , k_1 , and k_2 along with substitution which occurs due to the nonlinear S-Boxes. Let us denote the S-Box mapping by S , and thus an input x is transformed to an output which is $S[x]$. The mapping we choose for illustration is that of the block cipher PRESENT[10]. For convenience we present the mapping in Table 2.1.

From the notion of differentials we know that ex-oring with the key has no effect on the differential. However, the S-Box being a nonlinear layer prevents passing of the ex-or differential. Thus, the attacker can guess k_2 and obtain the value of x , for two different encryptions. We denote this by saying that c_0 gives x_0 , while c_1 gives x_1 . One can perform the inverse S-Boxes (which have to be invertible!), and obtain w_0 and w_1 . However, the uncertainty of k_1 prevents us from computing the values of v_0 and v_1 . However, we know $\Delta(w) = w_0 \oplus w_1 = v_0 \oplus v_1 = \Delta(v)$. Likewise, if the attacker chooses $\Delta(m) = m_0 \oplus m_1$, she can compute $\Delta(u)$, however the S-Box prevents from determining $\Delta(v)$.

So, differential cryptanalysis performs a differential analysis of the S-Box. Consider, the Table 2.2, where the input differential $i \oplus j$, for two inputs i and j , is chosen to be F. The output differential, $S[i] \oplus S[j]$ is computed, and its frequency is observed. It may be seen that some values do not occur in the output differential. Good design practice in the PRESENT S-Box ensures that the probability distribution is uniform, meaning the differentials, namely E, 4, 1, and F which occurs in the output occurs with the same probability, i.e., $\frac{1}{4}$. The attacker can thus choose the values of m_0 and m_1 , such that $\Delta(m) = F$, and thus $\Delta(u) = F$. Due to the differential property of the S-Box we know that in 4 out of 16 cases, the output differentials can be E, 4, 1, and F. Thus, $\Delta(w)$ is also any one of the above choices. The attacker also guesses k_2 , and obtains the value of $\Delta(w)$ by inverting the S-Box on the values x_0 and x_1 , obtained by guessing the key k_3 . It is clear that those guesses which provide $\Delta(w)$ any other value, but E, 4, 1, F, can be eliminated as wrong keys. However, the differential property of the S-Box cannot distinguish the keys which lead to $\Delta(w)$

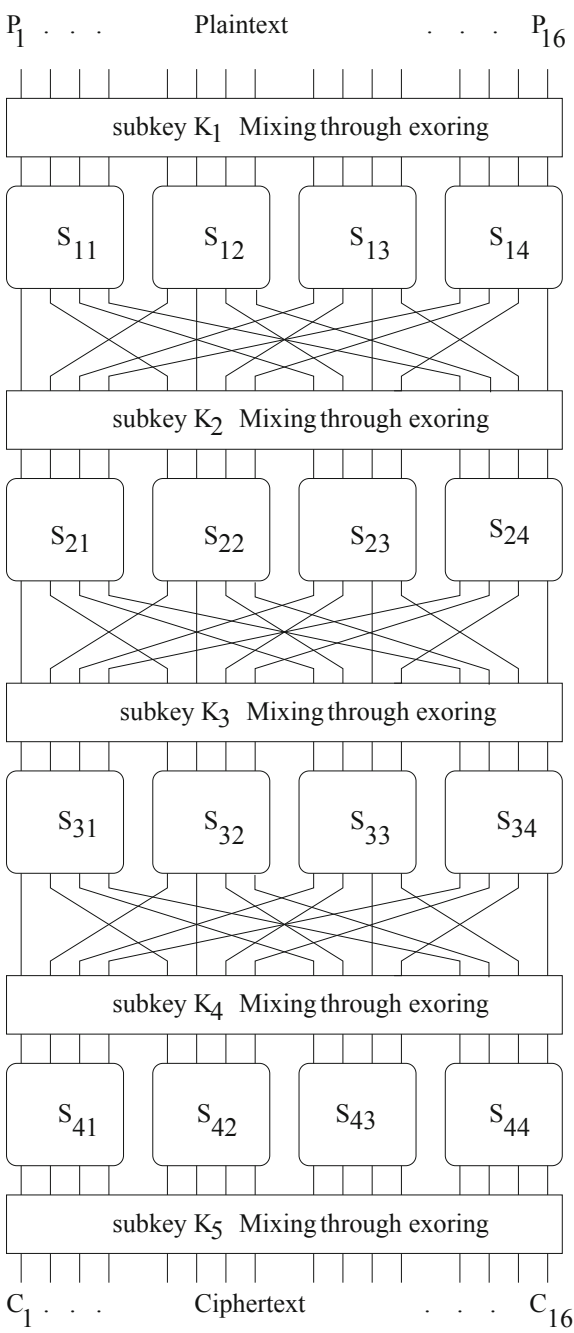
Table 2.2 Differential analysis of the S-Box

i	j	$S[i]$	$S[j]$	$S[i] \oplus S[j]$
0	F	C	2	E
1	E	5	1	4
2	D	6	7	1
3	C	B	4	F
4	B	9	8	1
5	A	0	F	F
6	9	A	E	4
7	8	D	3	E
8	7	3	D	E
9	6	E	A	4
A	5	F	0	F
B	4	8	9	1
C	3	4	B	F
D	2	7	6	1
E	1	1	5	4
F	0	2	C	E

equal to the above four values. This approximately eliminates 3/4 of the keys! This simple example shows the power of differential cryptanalysis, and can be used to distinguish the wrong keys from the correct ones.

In real life the block ciphers are cascades of substitution and permutations, and thus often referred to as SPN (Substitution Permutation Network) ciphers. However, the above discussion of differential cryptanalysis can be conceptually applied even for such a construction. In such a case, for a four-round cipher as shown in Fig. 2.9, an input–output differential is found for three rounds of the cipher which occurs with a very high probability. It is often defined as a differential trail, $\Delta_i \rightarrow \Delta_o$, where Δ_i and Δ_o are the input differential and output differential after the third round. The output differential also should ensure it is of low weight, meaning that it disturbs or affects less number of S-Boxes of the final round. The attacker can then guess a part of the last round key, corresponding to the disturbed S-Boxes, and then decrypt a portion of the ciphertext to check the differential at the output of the third round. It is expected that for the correct key the differential should be equal to Δ_o , with a very high probability. It may be noted that the efficiency of such an attack compared to a brute force attack is because it is a divide-and-conquer strategy, and thus can be successful by guessing portions of the key.

Fig. 2.9 The SPN block cipher



2.4 Asymmetric-Key Ciphers

The book also deals with attacks on asymmetric-key ciphers also known as public-key ciphers. These algorithms are often computation intensive and operate on large finite fields. For efficient design several field operation algorithms have been developed for supporting multiplication, inverse, exponentiation etc., and have been implemented in software libraries. The next section provides a quick overview on some of the commonly known techniques which are employed to realize the ciphers, and are hence targeted for demonstrating timing attacks. We focus on RSA, as it is still one of the most popular and utilized public-key algorithms. Further more, many of the attack techniques that we discuss in the book can be extended to other well known public-key algorithms, like elliptic curve cryptosystems.

2.5 RSA: An Asymmetric-Key Algorithm

RSA works by considering two keys: a *public key* is known to every one whereas a *private key* is *secret*. Encryption of a message is performed using the public key, but decryption requires the knowledge of the private key. All the operations are done mod n , where n is the product of two large distinct prime numbers p and q . The values of p and q are, however, private and hence not disclosed to all. The encryption key, which is public is a value b , where $1 \leq b \leq \phi(n)$ and $\phi(n)$ is the Euler's totient function. Very simply put, Euler's totient function or phi function, $\phi(n)$, is an arithmetic function that counts the positive integers less than or equal to n that are relatively prime to n . The decryption key is a private value a , which is selected such that $ab \equiv 1 \pmod{\phi(n)}$. The owner of the private key (p, q, a) publishes the value (b, n) , which is the public key.

The encryptor chooses a message x , where $x \in Z_n$. It may be mentioned that $Z_n = \{0, 1, \dots, n-1\}$. The encryption process is computing the cipher as $y \equiv x^b \pmod{n}$ using the public key b . Since the decryptor knows the value of a , which is the private key, he computes the value of x from y by computing $y^a \equiv (x^b)^a \pmod{n} \equiv x \pmod{n}$.

Algorithm 2.3: Square and Multiply algorithm for Exponentiation

Input: Ciphertext: C , Modulus: N and Private key: $k = (k_{m-1}, k_{m-2} \dots k_0)$, where $k_{m-1} = 1$

Output: Plaintext: $M \equiv C^k \pmod{N}$

```

1  $R = C$ 
2 for  $i = m-2$  to 0 do
3    $R = R^2 \pmod{N}$ 
4   if  $k_i = 1$  then
5      $R = R \times C \pmod{N}$ 
6   end
7 end
8 return  $R$ 
```

The security of *RSA* is based on the assumption that decryption can be performed only by the knowledge of the private key b . However, to obtain the private information from the public value a requires one to compute the modular inverse of a modulo $\phi(n)$. It is believed that to obtain $\phi(n)$ from n requires the knowledge of the prime factors of n , namely p and q . The security of *RSA* is thus based on the hardness assumption of factorization of large n . Thus, the underlying operation to perform *RSA* encryption and decryption is modular exponentiation, i.e., $y \equiv x^b \bmod n$, where b is typically 1024 bits or 2048 bits large. This is achieved by the popular square and multiply algorithm.

2.5.1 Square and Multiply Algorithm to Perform Exponentiation

In this section, we present the square and multiply algorithm to perform modular exponentiation. We present the decryption algorithm, which uses the decryption key, denoted as $k = a$. We focus on the decryption algorithm as that is a natural target for an attacker, as the secret key is involved in the computation. The key is an m -bit key, denoted as $k = (k_{m-1}, k_{m-2} \dots k_0)$, where $k_{m-1} = 1$.

This algorithm, as we later elaborate in the book, is naturally vulnerable against *SCA*. The reason being that the operations, namely squaring and multiplication, have different fingerprints on the side-channel leakage. For example, with respect to a timing attack, both requires different number of clock cycles to execute. We can observe that if a key bit is one, then a multiplication is performed, else not. The attacker hence tries to exploit this conditional property on the key bits to devise an attack. In the literature, there are several ways of performing a square and multiply algorithm to achieve exponentiation. One of the most popular techniques is called Montgomery Ladder, which is explained in the Algorithm 2.4.

It may be observed that in the Montgomery's Ladder, irrespective of the key bit, a multiplication and squaring is always performed. Thus, the design is naturally resistant against some *SCA*, which are possible over the naïve square and multiply algorithm.

Algorithm 2.4: Montgomery's Ladder for Exponentiation

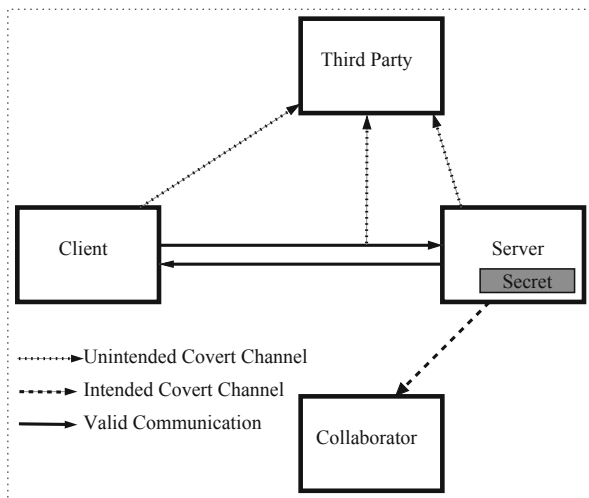
Input: Ciphertext: C , Modulus: N and Private key: $k = (k_{m-1}, k_{m-2} \dots k_0)$, where $k_{m-1} = 1$

Output: Plaintext: $M \equiv C^k \bmod N$

```

1  $P_1 = C, P_2 = C^2 \bmod N$ 
2 for  $i = m - 2$  to 0 do
3   if  $k_i = 1$  then
4      $P_1 = P_1 \times P_2 \bmod N, P_2 = P_2^2 \bmod N$ 
5   end
6   else
7      $P_2 = P_1 \times P_2 \bmod N, P_1 = P_1^2 \bmod N$ 
8   end
9 end
10 return  $P_1$ 
```

Fig. 2.10 Lampson's confinement problem



2.6 Confinement Problem and Covert Channels

Now that we have reviewed the essential components in cryptography, we move to side channels and how they covertly leak information about an executing application.

Consider a client using the services of a server. The client provides an input which is operated on by the server using some stored secret. In an untrusted environment, the client should ensure that the server does not communicate its input to a collaborator (the attacker), while the secret stored in the server should not be revealed to the client. Additionally no third-party should gain any information about the transactions between the client and server. This is the *confinement problem* as defined by Lampson in 1973 [10] and pictorially represented in Fig. 2.10.

Generally, the system can ensure that the server does not communicate with the collaborator by disabling writing to files, shared memories, and other inter-process communication (IPC) protocols. The server's secret can be protected by implementing memory protection using schemes such as paging and access control. In spite of these protection schemes, there still exists indirect paths by which data can be communicated. These indirect paths are not meant for communication, hence known as *covert channels*. They are of interest in the domain of computer security because they allow programs to bypass security policies of the system. They are all the more relevant in the cloud computing environments where such untrusting parties sharing resources are common. Covert channels can either be *intended* (where a nexus exists between the server and the collaborator) or *unintended* (where a third party obtains information about the computations). Covert channels are generally noisy, but information theory can be used to devise an encoding, which will allow data to get through reliably no matter how small the signal is, provided it is not zero [10].

An example of a covert channel was illustrated by Schaefer et al. in [11]. Consider the server and collaborator sharing a CPU. The client can signal information to the

collaborator by the amount of time it holds the CPU. For instance, holding the CPU for 10, 20, or 30 μ s can be used to represent a 0, 1, or 2 respectively [11]. This forms an *intentional* covert channel between the two cooperating processes (the client and the collaborator). Over the years several covert channels have been discovered such as the rate at which a program performs paging [10, 12], CPU scheduling [13], disk scheduling [13, 14], and cache memory usage [15, 16].

Covert channels do not always have to be intentional. There can also be unintentional covert channels, which results in transfer of information about a program execution to a third party. The program is unaware of these transfers. Such unintentional covert channels occur due to physical attributes of a device such as the power consumption and electromagnetic radiation, as well as execution time. These unintentional covert channels are commonly known as *side channels*. In 1996, Kocher showed that a timing side channel can be used to reveal the secret key of a cipher to an adversary [17]. Later, power consumption of the device was used for the same purpose [18]. These works that came to be known as SCA (side-channel attacks), kindled interest in the academic community and led to a new domain of cryptographic research; targeting implementations of ciphers rather than just the algorithm. The next section surveys SCA against ciphers that use unintentional covert timing channels. This class of attacks is called *timing attacks*.

2.7 Formal Analysis of Side-Channel Attacks

Since Shannon's benchmark paper in [19], there has been significant progress in the mathematical treatment given to cryptography. Ciphers, such as the three discussed in the previous section, underwent rigorous analysis with strong attack models such as adversaries who know the cipher algorithm, the input, as well as the output. The only secret being the cipher's key. All ciphers standardized and in use today have bounded security against these attack models.

The advent of SCA in [17] introduced a stronger (yet practical) attack model. Here, the adversary has access to side-channel information leakage of the encrypting device in addition to the input, output, and cipher algorithm. The mathematical tools developed so far failed to model these attack classes. It was not until 2004 that new models were developed by Micali and Reyzin to analyze cryptography in the presence of side-channel leakage [20]. This they called *physical observable cryptography*. The new theory was enhanced by the works of Standaert [21–24] and by Backes and Kopf [25, 26, 27] and used to provide a fair evaluation and comparison of ciphers in the presence of side-channel leakage. This has led to the development of a new class of cryptographic algorithms and countermeasures with provable security in the presence of leakages (for example [28, 29, 30]). The algorithms currently available are inefficient to implement. The hope is that the tools and models will lead to practically realizable ciphers with provable resistance against SCA. In this section we present briefly the formal notions in side-channel analysis.

Let $E_{\mathbf{k}}$ be a cipher having a secret key \mathbf{k} chosen uniformly from the set \mathcal{K}^m for some positive integer m . Let $\tilde{E}_{\mathbf{k}}$ be an implementation of $E_{\mathbf{k}}$ on a device. A *q-limited side-channel key recovery adversary* is a statistical program, which can make at-most q queries to $\tilde{E}_{\mathbf{k}}$ and monitor the leakage through channels such as power-consumption, timing, or electromagnetic radiation of the device. To quantify leakage, a *leakage function* $L(\cdot)$ is defined that mathematically abstracts the characteristics of the side channel and the measurement setup. For example, it is well-known that power consumption can be modeled in terms of Hamming weight or Hamming distances for CMOS devices.

The q -limited side-channel key recovery adversary follows a divide-and-conquer strategy by splitting \mathbf{k} into smaller parts for example $\mathbf{k} = (k_1 \| k_2 \| k_3 \| \dots \| k_m)$, where $k_1, k_2, \dots, k_m \in \mathcal{K}$, and each key part is targeted independently. Further, the leakage partitions the key space \mathcal{K} into equivalence classes such that the SCA cannot distinguish between two keys in the same class. The goal of the adversary is to guess a key class with nonnegligible probability. To do so, typically the attack has two phases. An online phase in which the q side-channel leakages are collected and an offline phase in which the leakages are analyzed using statistical distinguishers in order to obtain a ranking of the keys in \mathcal{K} in an order of their likelihood.

There are two metrics by which the success of the attack can be measured. The first defines the *o*th order *success rate* as the probability that the correct key is ranked among the top o keys. For example, if $\mathcal{G} = (g_1, g_2, \dots, g_o, g_{o+1}, \dots, g_{|\mathcal{K}|})$ is the ordered sequence of keys ranked from most likely to least likely, then the probability that the correct value of the key is in g_1, g_2, \dots, g_o is the *o*th order success rate of the attack. Alternatively, *guessing entropy* [31] can be used as a metric to evaluate the success of an attack. This measures the average number of guesses that are required before obtaining the correct key. For example, if the correct key is present in the j th location in the ranking \mathcal{G} then the guessing entropy is j . The second metric uses information-theoretic metrics to determine for example $\mathbb{H}[K|L]$, i.e., the entropy of the key given the leakage. This should be much less than $\mathbb{H}[K]$ for a strong attack.

This book shows how timing attacks can be modeled by analyzing the cipher's memory access patterns. The mathematical framework thus developed is used to evaluate ciphers for their resistance against timing attacks and choose implementation strategies for the ciphers.

2.8 Conclusion

This chapter provided an overview of various symmetric and asymmetric encryption schemes. Classical cryptanalytic techniques were dwelt upon, and side channels and their formal analysis were introduced. To a side-channel attacker, it is not just the enciphering algorithm and their implementations that are important, but the system and CPU architecture is also crucial. The next chapter provides an overview of modern CPUs and the few components in them that have been used to develop SCA.

References

1. Gueron S (2010) Intel Advanced Encryption Standard (AES) instructions set (Rev : 3.0)
2. Zheng Y, Matsumoto T, Imai H (1989) On the construction of block ciphers provably secure and not relying on any unproved hypotheses. In: Brassard G (ed) CRYPTO. Lecture notes in computer science, vol 435. Springer, Berlin, pp 461–480
3. Federal Information Processing Standards Publication 197 (2001) Announcing the Advanced Encryption Standard (AES)
4. Stinson D (2002) Cryptography: theory and practice, 2nd edn. Chapman and Hall, London, pp 117–154
5. Daemen J, Rijmen V (2002) The design of Rijndael: AES—the Advanced Encryption Standard. Springer, Berlin
6. Shirai T, Shibutani K, Akishita T, Moriai S, Iwata T (2007) The 128-bit blockcipher CLEFIA (extended abstract). In: Biryukov A (ed) FSE Lecture notes in computer science, vol 4593. Springer, Berlin, pp 181–195
7. Sony Corporation (2007) The 128-bit blockcipher CLEFIA : algorithm specification
8. Aoki K, Ichikawa T, Kanda M, Matsui M, Moriai S, Nakajima J, Tokita T (2000) Camellia: A 128-bit block cipher suitable for multiple platforms—design and analysis. In: Stinson DR, Tavares SE (eds) Selected areas in cryptography. Lecture notes in computer science, vol 2012. Springer, Berlin, pp 39–56
9. Knudsen LR, Robshaw MJB (2011) The block cipher companion. Springer, Berlin
10. Lampson BW (1973) A note on the confinement problem. Commun ACM 16(10):613–615. <http://doi.acm.org/10.1145/362375.362389>
11. Schaefer M, Gold B, Linde R, Scheid J (1977) Program confinement in KVM/370. In: Proceedings of the 1977 annual conference, ser. ACM '77. ACM: New York, pp. 404–410. <http://doi.acm.org/10.1145/800179.1124633>. Accessed Dec 2013
12. Van Vleck T (1990) Timing channels. Multics, technical report, 1990
13. Kemmerer RA (1983) Shared resource matrix methodology: an approach to identifying storage and timing channels. ACM Trans Comput Syst 1(3):256–277. <http://doi.acm.org/10.1145/357369.357374>
14. Karger PA, Wray JC (1991) Storage channels in disk arm optimization. IEEE symposium on security and privacy. IEEE, Oakland, pp 52–63
15. Wray JC (1991) An analysis of covert timing channels. In: Research in security and privacy, 1991. Proceedings, 1991 IEEE computer society symposium. May 1991, pp 2–7
16. Hu W-M (1992) Lattice scheduling and covert channels. In: Proceedings of the 1992 IEEE symposium on security and privacy. SP '92. IEEE Computer Society, Washington, DC, pp 52–61
17. Kocher PC (1996) Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz N (ed) CRYPTO '96: proceedings of the 16th annual international cryptography conference on advances in cryptology. Lecture notes in computer science, vol 1109. Springer-Verlag, London, pp 104–113
18. Kocher PC, Jaffe J, Jun B (1999) Differential power analysis. In: Wiener MJ (ed) CRYPTO. Lecture notes in computer science, vol 1666. Springer, Berlin, pp 388–397
19. Shannon CE (1949) Communication theory of secrecy systems. Bell Syst Tech J 28:656–715
20. Micali S, Reyzin L (2004) Physically observable cryptography (extended abstract). In: Naor M (ed) TCC. Lecture notes in computer science, vol 2951. Springer, Berlin, pp 278–296
21. Standaert F-X, Peeters E, Archambeau C, Quisquater J-J (2006) Towards security limits in side-channel attacks. In: Goubin L, Matsui M (eds) CHES. Lecture notes in computer science, vol 4249. Springer, Berlin, pp 30–45
22. Standaert F-X, Pereira O, Yu Y, Quisquater J-J, Yung M, Oswald E (2009c) Leakage resilient cryptography in practice. Cryptology ePrint archive, Report 2009/341. <http://eprint.iacr.org/>. Accessed June 2010

23. Standaert F-X, Malkin T, Yung M (2009b) A unified framework for the analysis of side-channel key recovery attacks. In: Joux A (ed) EUROCRYPT. Lecture notes in computer science, vol 5479. Springer, Berlin, pp 443–461
24. Standaert F-X, Koeune F, Schindler W (2009a) How to compare profiled side-channel attacks? In: Abdalla M, Pointcheval D, Fouque P-A, Vergnaud D (eds) ACNS. Lecture notes in computer science, vol 5536, pp 485–498
25. Köpf B, Basin DA (2007) An information-theoretic model for adaptive side-channel attacks. In: Ning P, di Vimercati SDC, Syverson PF (eds) ACM conference on computer and communications security. ACM, Alexandria, pp 286–296
26. Backes M, Köpf B (2008) Formally bounding the side-channel leakage in unknown-message attacks. In: Jajodia S, ópez JL (eds) ESORICS. Lecture notes in computer science, vol 5283. Springer, Berlin, pp 517–532
27. Backes M, Köpf B, Rybalchenko A (2009) Automatic discovery and quantification of information leaks. In: IEEE symposium on security and privacy. IEEE Computer Society, 2009, pp 141–153
28. Dziembowski S, Pietrzak K (2008) Leakage-resilient cryptography. In: FOCS. IEEE Computer Society, 2008, pp 293–302
29. Naor M, Segev G (2009) Public-key cryptosystems resilient to key leakage. In: Halevi S (ed) CRYPTO. Lecture notes in computer science, vol 5677. Springer, Berlin, pp 18–35
30. Pietrzak K (2009) A leakage-resilient mode of operation. In: Joux A (ed) EUROCRYPT. Lecture notes in computer science, vol 5479. Springer, Berlin, pp 462–482
31. Massey J (1994) Guessing and entropy. In: Information theory, 1994. Proceedings, 1994 IEEE international symposium, 1994, p 204

Timing Channels in Cryptography

A Micro-Architectural Perspective

Rebeiro, C.; Mukhopadhyay, D.; Bhattacharya, S.

2015, XVII, 152 p. 75 illus., 14 illus. in color., Hardcover

ISBN: 978-3-319-12369-1