CHAPTER 9

Fault Detection in Cryptographic Systems

Cryptographic algorithms are being applied in an increasing number of devices to satisfy their high security requirements. Many of these devices require high-speed operation and include specialized hardware encryption and/or decryption circuits for the selected cryptographic algorithm. A unique characteristic of these circuits is their very high sensitivity to faults. Unlike ordinary arithmetic/logic circuits such as adders and multipliers, even a single data bit fault in an encryption or decryption circuit will, in most cases, spread quickly and result in a totally scrambled output (an almost random pattern). There is, therefore, a need to prevent such faults or, at the minimum, be able to detect them.

There is another, even more compelling, reason for paying special attention to fault detection in cryptographic devices. The cryptographic algorithms (also called ciphers) that are being implemented are designed so that they are difficult to break. To obtain the secret key, which allows the decryption of encrypted information, an attacker must perform a prohibitively large number of experiments. However, it has been shown that by deliberately injecting faults into a cryptographic device and observing the corresponding outputs, the number of experiments needed to obtain the secret key can be drastically reduced. Thus, incorporating some form of fault detection into cryptographic devices is necessary for security purposes as well as for data integrity.

We start this chapter with a brief overview of two important classes of ciphers, namely, symmetric key and asymmetric (or public) key, and describe the fault injection attacks that can be mounted against them. We then present techniques that can be used to detect the injected faults in an attempt to foil the attacks.

9.1 Overview of Ciphers

Cryptographic algorithms use secret keys for encrypting the given data (known as *plaintext*) thus generating a *ciphertext*, and for decrypting the ciphertext to reconstruct the original plaintext. The keys used for the encryption and decryption steps can be either identical (or trivially related), leading to what are known as *symmetric key* ciphers, or different, leading to what are known as *asymmetric key* (or *public key*) ciphers. Symmetric key ciphers have simpler, and therefore faster, encryption and decryption processes compared with those of asymmetric key ciphers. The main weakness of symmetric key ciphers is the shared secret key, which may be subject to discovery by an adversary and must therefore be changed periodically. The generation of new keys, commonly carried out using a pseudo-random-number generator (see Section 10.4), must be very carefully executed because, unless properly initialized, such generators may result in easy to discover keys. The new keys must then be distributed securely, preferably by using a more secure (but also more computationally intensive) asymmetric key cipher.

9.1.1 Symmetric Key Ciphers

Symmetric key ciphers can be either *block ciphers*, which encrypt a block of a fixed number of plaintext bits at the same time, or *stream ciphers*, which encrypt 1 bit at a time. Block ciphers are more commonly used, and are therefore the focus of this chapter.

Some well-known block cyphers include the Data Encryption Standard (DES) and the more recent Advanced Encryption Standard (AES). DES uses 64-bit plaintext blocks and a 56-bit key, whereas AES uses 128-bit blocks and keys of size between 128 and 196 bits. Longer secret keys are obviously more secure, but the size of the data block also plays a role in the security of the cipher. For example, smaller blocks may allow frequency-based attacks, such as relying on the higher frequency of the letter “e” in English-language text.

Almost all symmetric key ciphers use the same key for encryption and for decryption. The process used for encryption must be reversible so that the reverse process followed during decryption can generate the original plaintext. The main objective of the encryption process is to scramble the plaintext as much as possible. This is done by repeating a computationally simple series of steps (called a *round*) several times to achieve the desired scrambling.

The DES cipher follows the approach ascribed to Feistel. The Feistel scheme divides the block of plaintext bits into two parts B_1 and B_2. B_1 is unchanged, whereas the bits in B_2 are added (using modulo-2 addition, which is the logical bit-wise Exclusive-OR (XOR) operation) to a one-way hash function $F(B_1, K)$, where K is the key. A hash function is a function that takes a long input string (in general, of any length) and produces a fixed-length output string. A function is called a one-way hash function if it is hard to reverse the process and find an input string that will generate a given output value. The two subblocks B_1 and $B_2 + F(B_1, K)$ are then swapped.

These operations constitute a round, and the round is repeated several times. Following a round, we end up with $B_1^{'} = B_2 + F(B_1, K)$ and $B_2^{'} = B_1$. A single round is not secure since the bits of B_1 are unchanged and were only moved, but repeating the round several times will considerably scramble the original plaintext.

The one-way hash function F may seem to prevent decryption. Still, by the end of the round, both B_1 and the key K are available and it is possible to recalculate $F(B_1, K)$ and thus obtain B_2. Therefore, all the rounds can be "undone" in reverse order to retrieve the plaintext.

DES has been the first official standard cipher for commercial purposes. It became a standard in 1976, and although there is currently a newer standard (AES established in 2002), the use of DES is still widespread either in its original form or in its more secure variation called Triple DES. Triple DES applies DES three times with different keys and offers as a result a higher level of security (one variation uses three different keys for a total of 168 bits instead of 56 bits, while another variation uses 112 bits).

The Feistel-function–based structure of DES is shown in Figure 9.1. It consists of 16 identical rounds similar to the one described above. Each round first uses a Feistel function (the F block in the figure), performs the modulo-2 addition (the $\oplus$ circle in the figure), and then swaps the two halves. In addition, DES includes an initial and final permutations (see Figure 9.1) that are inverses and cancel each other. These do not provide any additional scrambling and were included to simplify loading blocks of data in the original hardware implementation.

The 16 rounds use different 48-bit subkeys generated by a key schedule process shown in Figure 9.2. The original key has 64 bits, eight of which are parity bits, so the first step in the key schedule (the "*Permuted Choice 1*" in Figure 9.2) is to select 56 out of the 64 bits. The remaining 16 steps are similar: the 56 incoming bits are split into two 28-bit halves, and each half is rotated to the left by either one or two bits (specified for each step). Then, 24 bits from each half are selected by the "*Permuted Choice 2*" block to generate the 48-bit round subkey. As a result of the rotations, performed by the "<<<" block in the figure, a different set of bits is used in each subkey.

The particular Feistel (hash) function used in DES is shown in Figure 9.3. It consists of four steps:

1. *Expansion.* The 32 input bits are expanded to 48, using an expansion permutation that duplicates some of the bits.
2. *Adding a key.* The 48-bit result is added (addition modulo-2 which is a bit-wise XOR operation) to a 48-bit subkey generated by the key schedule process.

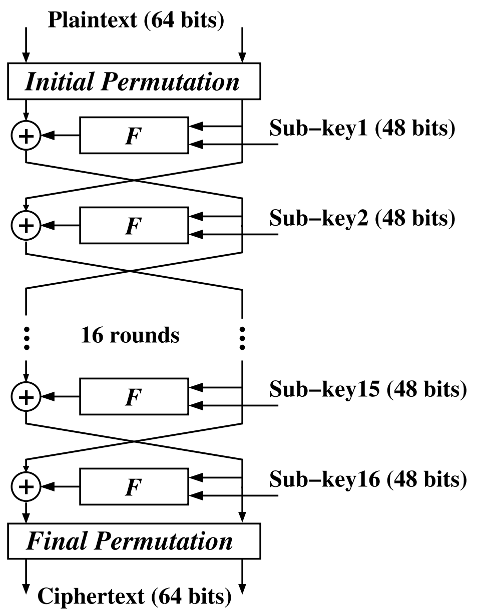

FIGURE 9.1 The overall structure of the data encryption standard (DES).

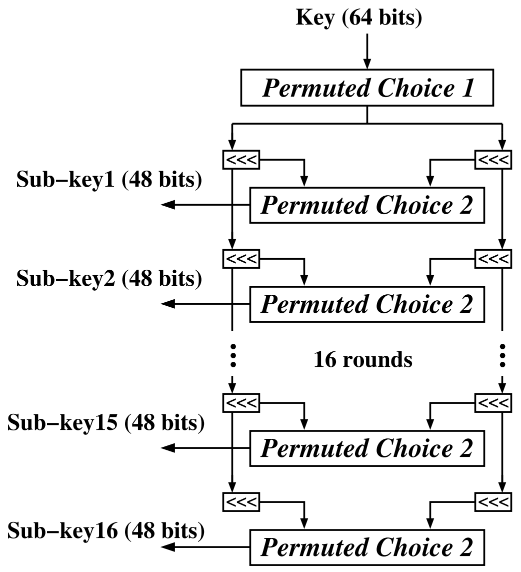

FIGURE 9.2 The key schedule process for DES.

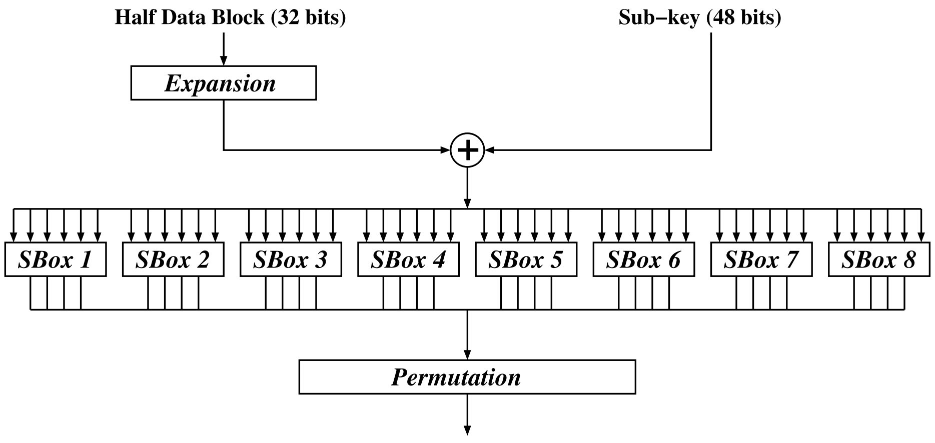

FIGURE 9.3 The Feistel function in DES.

3. *Substitution.* The 48-bit result of step 2 is divided into eight groups of 6 bits each, which are then processed by substitution boxes (called *SBoxes*). An SBox generates 4 bits according to a nonlinear transformation implemented as a lookup table.
4. *Permutation.* The 32 bits generated by the eight SBoxes undergo a permutation.

Two crucial properties that every good cipher must have are called *confusion* and *diffusion*. Confusion refers to establishing a complex relationship between the ciphertext and the key, and diffusion implies that any natural redundancy that exists in the plaintext (and can be exploited by an adversary) will dissipate in the ciphertext. In DES, most of the *confusion* is provided by the SBoxes, and the expansion and permutation provide the *diffusion*. If the confusion and diffusion are done correctly, a single bit change in the plaintext will cause every bit of the ciphertext to change with a probability of 0.5, independently of the others.

In 1999, a specially designed circuit was successful in breaking a DES key in less than 24 hours, demonstrating that the security provided by the 56-bit key is weak. Consequently, Triple DES has been declared as the preferred cipher and was itself later replaced in 2002 by AES, described next.

AES does not use a Feistel function; instead, it is based on substitutions and permutations, with most of its calculations being finite-field operations. AES uses blocks of 128-bit plaintext and three possible key sizes of 128, 192, or 256 bits. The 128-bit block is represented as a 4×4 array of bytes called the *state*, which is denoted by S with byte elements $s_{i,j}$ $(0 \leqslant i,j \leqslant 3)$. The state S is modified during each encryption round, until the final ciphertext is produced. Each round of the encryption process consists of four steps (see Figure 9.4):

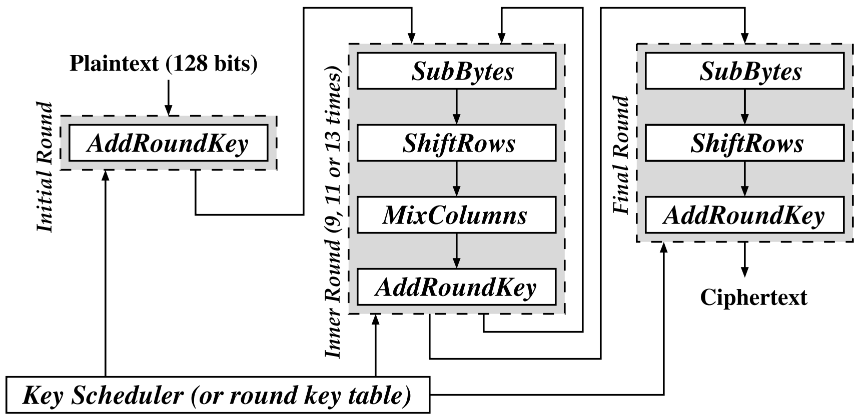

FIGURE 9.4 The overall structure of the advanced encryption standard (AES).

TABLE 9-1 ■ The advanced encryption standard (AES) SBox: substitution values for the byte xy (in hexadecimal format)

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

1. *SubBytes.* Each byte in the state matrix undergoes (independently of all other bytes) a nonlinear substitution of the form $T(s_{i,j}^{-1})$. Due to the complexity of this transformation, its 256 possible outcomes are (in almost all implementations of AES) precomputed and stored in an SBox lookup table. Unlike in DES, this is an 8- to 8-bit substitution (shown in Table 9-1) rather than a 6- to 4-bit one. The AES SBox has been designed to resist simple attacks.

2. *ShiftRows.* The bytes of the first, second, third, and fourth rows of the state matrix are rotated by 0, 1, 2, and 3 bytes, respectively. The state after this step is

$$S = \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,1} & s_{1,2} & s_{1,3} & s_{1,0} \\ s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3,3} & s_{3,0} & s_{3,1} & s_{3,2} \end{bmatrix} \tag{9.1}$$

so that every column of the matrix is now composed of bytes from all columns of the input matrix.

3. *MixColumns.* The four bytes in each column are used to generate four new bytes through linear transformations, as follows ($j = 0,1,2,3$)

$$\begin{aligned} s_{0,j} &= (\alpha \otimes s_{0,j}) \oplus (\beta \otimes s_{1,j}) \oplus s_{2,j} \oplus s_{3,j} \\ s_{1,j} &= s_{0,j} \oplus (\alpha \otimes s_{1,j}) \oplus (\beta \otimes s_{2,j}) \oplus s_{3,j} \\ s_{2,j} &= s_{0,j} \oplus s_{1,j} \oplus (\alpha \otimes s_{2,j}) \oplus (\beta \otimes s_{3,j}) \\ s_{3,j} &= (\beta \otimes s_{0,j}) \oplus s_{1,j} \oplus s_{2,j} \oplus (\alpha \otimes s_{3,j}) \end{aligned} \tag{9.2}$$

where $\alpha = x$ (or 02 in hexadecimal notation), $\beta = x + 1$ (or 03 in hexadecimal notation). $\otimes$ and $\oplus$ are the modulo-2 multiply and add operations, respectively, of the polynomial representations of the state bytes, and the α and β coefficients. These operations are performed modulo the irreducible generator polynomial of AES, which is $g(x) = x^8 + x^4 + x^3 + x + 1$. Polynomial presentations of binary numbers and operations modulo a given generator polynomial have been discussed in Section 3.1. The MixColumns step together with ShiftRows provide the required diffusion in the AES cipher.

4. *AddRoundKey.* The round subkey is added (modulo-2) to the state. As in DES, separate round subkeys are generated using a key schedule process.

All four steps are performed in nine out of the 10 rounds of a 128-bit key implementation, but in the 10th round, the MixColumns step is omitted. In addition, prior to the first round, the first subkey is added to the original plaintext (see Figure 9.4). The round subkeys are either generated on-the-fly following the key schedule process shown in Figure 9.5 or are taken out of a lookup table that is filled up every time a new key is established. The total number of rounds is increased to 12 and 14 for a 192-bit key and a 256-bit key AES, respectively.

EXAMPLE

A detailed example to illustrate the use of the AES algorithm (or for that matter, any other symmetric key cipher such as DES) for even the smallest sizes of its parameters (number of bits in the key and plaintext) will be tedious and not very illuminating. We present therefore only some of the key steps of the ex-

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

FIGURE 9.5 The key schedule of AES ($Nr = 10, 12, 14$ is the number of rounds, $Nk = 4, 6, 8$ is the number of 32-bit words in the plaintext, and $Rcon$ is an array of round constants, $Rcon[j] = (x^{j-1}, 00, 00, 00)$).

$$\begin{bmatrix} 32 & 88 & 31 & e0 \\ 43 & 5a & 31 & 37 \\ f6 & 30 & 98 & 07 \\ a8 & 8d & a2 & 34 \end{bmatrix}$$

(a) Initial state matrix

$$\begin{bmatrix} 2b & 28 & ab & 09 \\ 7e & ae & f7 & cf \\ 15 & d2 & 15 & 4f \\ 16 & a6 & 88 & 3c \end{bmatrix}$$

(b) Key added in round 1

$$\begin{bmatrix} 19 & a0 & 9a & e9 \\ 3d & f4 & c6 & f8 \\ e3 & e2 & 8d & 48 \\ be & 2b & 2a & 08 \end{bmatrix}$$

(c) State matrix — end of round 1

$$\begin{bmatrix} d4 & e0 & b8 & 1e \\ 27 & bf & b4 & 41 \\ 11 & 98 & 5d & 52 \\ ae & f1 & e5 & 30 \end{bmatrix}$$

(d) After SubBytes

$$\begin{bmatrix} d4 & e0 & b8 & 1e \\ bf & b4 & 41 & 27 \\ 5d & 52 & 11 & 98 \\ 30 & ae & f1 & e5 \end{bmatrix}$$

(e) After ShiftRows

$$\begin{bmatrix} 04 & e0 & 48 & 28 \\ 66 & cb & f8 & 06 \\ 81 & 19 & d3 & 26 \\ e5 & 9a & 7a & 4c \end{bmatrix}$$

(f) After MixColumns

$$\begin{bmatrix} a0 & 88 & 23 & 2a \\ fa & 54 & a3 & 6c \\ fe & 2c & 39 & 76 \\ 17 & b1 & 39 & 05 \end{bmatrix}$$

(g) The key added in round 2

$$\begin{bmatrix} a4 & 68 & 6b & 02 \\ 9c & 9f & 5b & 6a \\ 7f & 35 & ea & 50 \\ f2 & 2b & 43 & 49 \end{bmatrix}$$

(h) State matrix — end of round 2

$$\begin{bmatrix} 39 & 02 & dc & 19 \\ 25 & dc & 11 & 6a \\ 84 & 09 & 85 & 0b \\ 1d & fb & 97 & 32 \end{bmatrix}$$

(i) State matrix — end of round 10

FIGURE 9.6 Example illustrating the AES algorithm.

ample that appears in full detail in the official AES document (see the Further Reading section).

Suppose the 128-bit plaintext is

$$32\,43\,f6\,a8\,88\,5a\,30\,8d\,31\,31\,98\,a2\,e0\,37\,07\,34$$

and the 128-bit key is

$$2b\,7e\,15\,16\,28\,ae\,d2\,a6\,ab\,f7\,15\,88\,09\,cf\,4f\,3c$$

Both have 32 hexadecimal digits and are shown in a matrix format in Figures 9.6a and b, respectively. The reader can verify that the byte-wise XOR operation of these two matrices yields the state matrix at the end of round 1, shown in Figure 9.6c.

The first step in round 2 is SubBytes and its results are shown in Figure 9.6d. For example, the first byte in the state matrix was $s_{0,0} = 19$, and based on the corresponding entry in Table 9-1, it is replaced by $d4$. The second step is ShiftRows, and Figure 9.6e shows the results of rotating the first, second, third, and fourth rows of the matrix by 0, 1, 2, and 3 bytes, respectively. The next step is MixColumns, and its results are shown in Figure 9.6f. For example, the first byte in the state matrix is calculated based on Equation 9.2 as follows:

$$\begin{aligned} s_{0,0} &= (\alpha \otimes s_{0,0}) \oplus (\beta \otimes s_{1,0}) \oplus s_{2,0} \oplus s_{3,0} \\ &= (02 \otimes d4) \oplus (03 \otimes bf) \oplus 5d \oplus 30 = 1b8 \oplus 1c1 \oplus 5d \oplus 30 = 04 \end{aligned}$$

Note that since the result is smaller than 100 (x^8 in polynomial notation), there is no need to further reduce it modulo-$g(x)$ (recall that $g(x) = x^8 + x^4 + x^3 + x + 1$ is the generator polynomial of AES).

The situation is different when calculating the second byte in the first column. Here,

$$\begin{aligned} s_{1,0} &= s_{0,0} \oplus (\alpha \otimes s_{1,0}) \oplus (\beta \otimes s_{2,0}) \oplus s_{3,0} \\ &= d4 \oplus (02 \otimes bf) \oplus (03 \otimes 5d) \oplus 30 = d4 \oplus 17e \oplus e7 \oplus 30 = 17d \end{aligned}$$

This value must be reduced modulo-$g(x)$, and since

$$x^8 \bmod g(x) = x^4 + x^3 + x + 1$$

we obtain

$$17d \bmod g(x) = 7d \oplus \left(x^4 + x^3 + x + 1\right) = 7d \oplus 1b = 66$$

which is the final value of the second byte in the first column in Figure 9.6f.

We now need to calculate a new round key using the procedure in Figure 9.5. The original key is first rewritten as the following four words:

$$\begin{aligned} &w[0] = 2b7e1516, &&w[1] = 28aed2a6 \\ &w[2] = abf71588, &&w[3] = 09cf4f3c \end{aligned}$$

To calculate $w[4]$ (the first column in the key matrix for round 2), we start with

$$temp = w[i-1] = w[3] = 09cf4f3c$$

Then, we rotate this word by 1 byte obtaining $cf4f3c09$. Next, we substitute each of the 4 bytes using the SubBytes transformation in Table 9-1, yielding $8a84eb01$. We then perform a bit-wise XOR operation with

$$Rcon[1] = \left(x^{1-1}, 00, 00, 00\right) = 01000000$$

obtaining $8b84eb01$. Finally, we calculate

$$\begin{aligned} w[i] &= w[i-4] \quad \text{xor} \quad temp = w[0] \quad \text{xor} \quad 8b84eb01 \\ &= 2b7e1516 \quad \text{xor} \quad 8b84eb01 = a0fafe17 \end{aligned}$$

This is the first column in the key matrix in Figure 9.6g. Adding the resulting key matrix to the state matrix, we obtain the new state matrix shown in Figure 9.6h. Continuing this process for the remaining rounds (recall that in the last round the MixColumns step is skipped) results in the ciphertext

$$39\,25\,84\,1d\,02\,dc\,09\,fb\,dc\,11\,85\,97\,19\,6a\,0b\,32$$

as shown in Figure 9.6i.

If a single bit is changed in the plaintext, for example, instead of

$$32\,43\,f6\,a8\,88\,5a\,30\,8d\,31\,31\,98\,a2\,e0\,37\,07\,34$$

we use

$$30\,43\,f6\,a8\,88\,5a\,30\,8d\,31\,31\,98\,a2\,e0\,37\,07\,34$$

a very different ciphertext is obtained:

$$c0\,06\,27\,d1\,8b\,d9\,e1\,19\,d5\,17\,6d\,bc\,ba\,73\,37\,c1$$

Similarly, if a single bit is changed in the key, for example, instead of

$$2b\,7e\,15\,16\,28\,ae\,d2\,a6\,ab\,f7\,15\,88\,09\,cf\,4f\,3c$$

we use

$$2a\,7e\,15\,16\,28\,ae\,d2\,a6\,ab\,f7\,15\,88\,09\,cf\,4f\,3c$$

the ciphertext produced is

$$c4\,61\,97\,9e\,e4\,4d\,e9\,7a\,ba\,52\,34\,8b\,39\,9d\,7f\,84$$

These two examples illustrate the fact that even a single-bit fault may result in a totally scrambled (almost random) output, demonstrating the significance of detecting such faults. ■

9.1.2 Public Key Ciphers

Unlike symmetric key ciphers, asymmetric key ciphers (also known as public key ciphers) allow users to communicate securely without having access to a shared secret key. Public key ciphers are, however, considerably more computationally complex than symmetric key ciphers. Instead of a single key shared by the two entities communicating with each other, the sender and recipient each have two cryptographic keys called the public key and the private key. The private key is kept secret, and the public key may be widely distributed. In a way, one of the two keys can be used to “lock” a safe, whereas the other key is needed to unlock it. If a sender encrypts a message using the recipient’s public key, only the recipient can decrypt it using the corresponding private key.

Another noteworthy application of public key ciphers is sender authentication: the sender encrypts a message with her own private key. By managing to decrypt the message using the sender’s public key, the recipient is assured that the sender (and no one else) generated the message.

The best-known public key cipher is the RSA algorithm named after its three inventors Rivest, Shamir and Adleman, but other public key ciphers have been developed and are in use. Person A wishing to use the RSA cipher must first generate a secret private key and a public key. The latter will be distributed to everyone who may wish to communicate with her. The key generation process consists of the following steps:

1. Select two large prime numbers p and q, and calculate their product $N = pq$.
2. Select a small odd integer e that is relatively prime to
$$\phi(N) = (p-1)(q-1)$$
Two numbers (not necessarily primes) are said to be relatively prime if their only common factor is 1. For example, 6 and 25 are relatively prime, although neither is a prime number.
3. Find the integer d that satisfies
$$de = 1 \bmod \phi(N)$$
(d is often called the “inverse” of e).

The pair (e, N) constitutes the public key, and A should broadcast it to everyone who may wish to communicate with her. The pair (d, N) will serve as A’s secret private key. The security provided by RSA depends on the difficulty of factoring the large integer N into its prime factors. Small integers can be factored in a reasonable amount of time, allowing the secret private key to be easily derived from the public key. To make the factoring time prohibitively large, each of the prime numbers p and q must have at least a hundred digits.

Given a message M that person B wishes to send to A, B will encrypt it using A's public key as

$$S = M^e \bmod N$$

Note that this encryption scheme makes it necessary to restrict the message M to

$$0 \leqslant M \leqslant N-1$$

Upon receiving the encrypted message S, A will decrypt it using her private key by calculating

$$S^d \bmod N = M^{de} \bmod N$$

which can be shown to be equal to the original plaintext message M. The encryption and decryption of RSA messages thus entail exponentiations modulo-N.

Although there are techniques for reducing the complexity of such modular exponentiation (e.g., Montgomery reduction), the complexity of encryption and decryption for the RSA cipher is still considerably higher than that for symmetric key ciphers.

■ EXAMPLE

To illustrate the use of the RSA algorithm, consider the following simple example. Suppose we select the prime numbers $p = 7$ and $q = 11$, yielding $N = 77$ and $\phi(N) = 60$. We can then select $e = 7$, which is obviously relatively prime with respect to $\phi(N)$. The pair $(e, n) = (7, 77)$ constitutes our public key. We search now for d that satisfies $7d = 1 \bmod 60$, and find $d = 43$ (since $7 \cdot 43 = 301 = 1 \bmod 60$). Suppose now that B wishes to send us the message $M = 9$. B encrypts it using the public key $(e, N) = (7, 77)$, which we have given him, obtaining $9^7 \bmod 77 = 4782969 \bmod 77 = 37$. We receive 37 and decrypt it using our private key by calculating $37^{43} \bmod 77$, revealing the plaintext 9. ■

9.2 Security Attacks Through Fault Injection

The level of security provided by the different ciphers has not been proved in an absolute sense, and all ciphers rely on the difficulty of finding the secret key directly and having to resort to exhaustive searches which may take a prohibitive amount of time. However, attacks on cryptographic systems have been developed which take advantage of side-channel information. This is information that can be obtained from the physical implementation of a cipher rather than through exploitation of some weakness of the cipher itself. One example of such side-channel information is the time needed to perform an encryption (or decryption), which in certain implementations may depend on the bits of the key. This allows the attacker to narrow down the range of values which need to be attempted. Another example is the amount of power consumed in various steps of the encryption process: the power consumption profile of certain implementations may depend on whether the bits of the key are 0 or 1.

Schemes to protect cryptosystems against such attacks have been developed. For example, a random number of instructions that do not perform any useful calculation can be injected into the code, scrambling the relationship between the bits in the key and the total time needed to complete the encryption (or decryption). These randomly-injected instructions can also help protect against power measurements-based attacks. Other countermeasures that have been followed include designs that have a data-independent delay or use dual-rail logic that consumes the same power independently of whether a particular bit is 1 or 0. Most such techniques incur delay and/or power penalties.

An important type of side-channel attacks, which is of particular interest to us in this book, relies on the intentional injection of faults into a hardware implementation of a cipher. Such attacks proved to be both easy to apply and very efficient; an attacker can guess the secret key after a very small number of fault injection experiments. This has been shown to be true for many types of ciphers, both symmetric and asymmetric.

The different techniques for injecting intentional faults into a cryptographic device include varying the supply voltage (generating a *spike*), varying the clock frequency (generating a *glitch*), overheating the device, or, as is more commonly done, exposing the device to intense light using either a camera flash or a more precise laser (or X-ray) beam.

Injecting a fault through a voltage spike or a clock glitch is likely to render a complete byte (or even several bytes) faulty, whereas the more precise laser or X-ray beams may be successful in inducing a single-bit fault. Fault-based attacks have been developed for both cases, and since most of these attacks induce transient faults, they allow the attacker to repeat her attempts multiple times until sufficient information is collected for extracting the secret key and even use the device after breaking the cipher.

A practical issue that must be considered when mounting a fault-based attack is the need for precise timing of the fault injection. To achieve the desired effect, the fault must be injected during a particular step of the encryption or decryption algorithm. This turns out to be achievable in practice by analyzing the power and/or electromagnetic profile of the cryptographic device.

We next describe briefly possible fault attacks on symmetric and asymmetric key ciphers.

9.2.1 Fault Attacks on Symmetric Key Ciphers

Various fault injection based attacks on DES have been described, two of which are presented next.

TABLE 9-2 ■ Fault attack on data encryption standard (DES)

DES Key	Output
$K_0 = xx\ xx\ xx\ xx\ xx\ xx\ xx\ xx$	S_0
$K_1 = xx\ xx\ xx\ xx\ xx\ xx\ xx\ 00$	S_1
$K_2 = xx\ xx\ xx\ xx\ xx\ xx\ 00\ 00$	S_2
$K_3 = xx\ xx\ xx\ xx\ xx\ 00\ 00\ 00$	S_3
$K_4 = xx\ xx\ xx\ xx\ 00\ 00\ 00\ 00$	S_4
$K_5 = xx\ xx\ xx\ 00\ 00\ 00\ 00\ 00$	S_5
$K_6 = xx\ xx\ 00\ 00\ 00\ 00\ 00\ 00$	S_6
$K_7 = xx\ 00\ 00\ 00\ 00\ 00\ 00\ 00$	S_7

In cryptographic devices that use DES (e.g., smart cards), the secret key is often stored in an EEPROM and then transferred to the memory when a message needs to be encrypted or decrypted. If the attacker can reset an entire byte of the key (set the eight bits of that byte to zero) during its transfer from the EEPROM to the memory, he can figure out the secret key. The attack consists of eight steps as outlined in Table 9-2. In all of these experiments, known (to the attacker) plaintext messages are encrypted with a different number of bytes of the key being forced to 0 as shown in Table 9-2. Based on the ciphertext S_7, the attacker can derive the first byte of the secret key by trying out all possible values of the first byte until the value that would produce S_7 is found. Since in DES each byte of the key includes a parity bit, at most 128 values need to be checked rather than 256. In a similar manner, the second byte of the key can be found based on S_6. This procedure is continued until all eight bytes of the secret key are discovered.

A second fault-based attack relies on causing an instruction to fail (most commonly using clock glitches). For example, if the loop variable controlling the number of times the basic round is executed is corrupted and, as a result, only one or two rounds are executed, the task of finding the secret key is greatly simplified.

This type of attack can also be mounted against a device that uses AES and implements the cipher via software. Fault injection attacks on AES that focus, for example, on a byte of either the round subkey or on the state in the last round of the encryption have also been developed. Some of these attacks have been applied in practice to smart cards, yielding the secret key after fewer than 300 experiments. References to the descriptions of these attacks are provided in the Further Reading section.

9.2.2 Fault Attacks on Public (Asymmetric) Key Ciphers

Unlike symmetric key ciphers for which both encryption and decryption processes are vulnerable to security attacks, for a public key cipher, only the decryption process may be subject to attacks attempting to extract the secret private key. One easily understood fault attack on the RSA decryption process assumes that the attacker can flip a randomly selected single bit of the private key d. Given an encrypted message S and its corresponding plaintext M, both of which are known to the attacker, he flips a random bit of d. If the ith bit of d, d_i, is flipped to produce its complement $\bar{d}_i$, the decryption device will generate an erroneous plaintext $\widehat{M}$ instead of M. The ratio between these two is

$$\frac{\widehat{M}}{M} = \frac{S^{2^i \bar{d}_i}}{S^{2^i d_i}} \bmod N$$

If this ratio is equal to $S^{2^i} \bmod N$ for some i, the attacker can conclude that $d_i = 0$. A ratio of $\frac{1}{S^{2^i}} \bmod N$ for some i implies that $d_i = 1$. Repeating this process will eventually provide all the bits of the secret private key d.

In a similar way, the bits of d can be obtained by flipping a bit in the ciphertext S, and even by flipping two (or more) bits simultaneously. Showing this is left as an exercise for the reader. This type of attack can therefore, be successful even if the attacker is unable to precisely flip a single bit.

EXAMPLE

Let us use the example discussed in Section 9.1.2 with $(e,N) = (7,77)$ as the public key and $d = 43$ (or in binary $d_5d_4d_3d_2d_1d_0 = 101011$) as the private key. Suppose the decryption device receives the ciphertext 37 and produces the plaintext $M = 9$ if no fault is injected, and the erroneous text $\widehat{M} = 67$ if a single bit fault is injected into d. We now search for i such that $9 = (67 \cdot 37^{2^i}) \bmod 77$. It is easy to verify that among the possible values of i, $i = 3$ is the one because

$$\left(67 \cdot 37^8\right) \bmod 77 = (67 \cdot 53) \bmod 77 = 9$$

Consequently, we deduce that $d_3 = 1$. ■

9.3 Countermeasures

We presented above only a small sample out of the large number of possible fault-based attacks that can be mounted against cryptographic devices. Due to the relative ease of applying these attacks, it is obvious that proper countermeasures must be taken in order to keep the devices secure. Any such countermeasure must first detect the fault, and then prevent the attacker from observing the output of the device after the fault has been injected. Either the output could be blocked (by producing a constant value such as all zeroes) or a random result generated, misleading the attacker. Clearly, the original design of the device must be modified to include any such countermeasure.

Two approaches can be followed when modifying the design of a cryptographic device to protect it against fault injection–based attacks. One relies on duplicating the encryption or decryption process (using either hardware or time redundancy) and comparing the two results. This approach assumes that the injected faults are transient and will not manifest themselves in exactly the same time in the two calculations. This approach is easy to apply but may, in certain situations, impose an overhead too high to be practical. The second approach is based on error-detection codes (see Section 3.1) that usually require a smaller overhead compared with brute-force duplication, although possibly at the cost of a lower fault coverage. Thus, a trade-off between the fault coverage and the hardware and/or time overhead should be expected.

9.3.1 Spatial and Temporal Duplication

Applying duplication to the encryption (or decryption) procedure is quite straightforward. Spatial duplication requires redundant hardware to allow independent calculations so that faults injected into one hardware unit do not affect (in the same way) the other unit(s). Temporal redundancy can be applied by reusing the same hardware unit or re-executing the same software program, assuming that the manifestation of the injected faults will change from one execution to the other. These schemes are similar to the conventional hardware and time redundancy techniques that are described in Chapter 2. The recalculation with shifted/modified operands techniques that have been described in Section 5.2.4 can be used here to prevent the possibility of both computations being affected by the injected fault in exactly the same way.

A different scheme for applying duplication relies on having a separate hardware unit or software program for executing the reverse procedure. For example, after completing the encryption, the decryption unit or program is applied to the ciphertext, and only if the result of the decryption is equal to the original plaintext is the ciphertext considered fault-free and is output.

The latter approach is costly if applied to an RSA decryption device. The decrypted result $\widehat{M}$ obtained from the received encrypted message S is verified by calculating $\hat{S} = \widehat{M}^e \bmod N$ and comparing $\hat{S}$ to S. This calculation is time-consuming if the public key e is very large.

9.3.2 Error-Detecting Codes

This section illustrates the use of error-detecting codes (EDCs) for detecting faults in the encryption process of symmetric key ciphers. Similar rules apply to using EDCs during the decryption and key schedule procedures, because these use the same basic mathematical operations as the encryption.

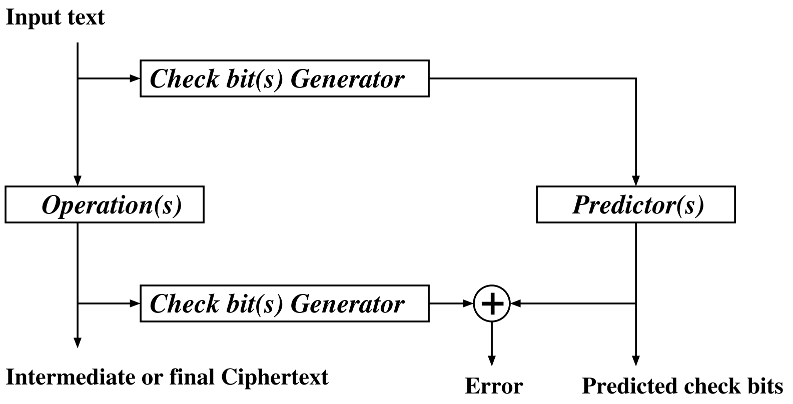

FIGURE 9.7 The general structure for detecting faults in encryption devices using error-detecting codes.

When using an EDC during the encryption process, check bits are first generated for the input plaintext, then for each operation(s) that the data bits undergo, the check bits of the expected result are predicted. Periodically, check bits for the actual result are generated and compared with the predicted check bits, and a fault is detected if the two sets do not match. The general approach is depicted in Figure 9.7. The validation checks can be scheduled at various granularities of the encryption, be it after every operation applied to the data, after each round, or only once at the end of the encryption process.

The first step, that of generating the check bits for the plaintext, is straightforward. The difficult part is devising the prediction rules for the new values of the check bits after each transformation that the data bits undergo during the encryption process. The complexity of these prediction rules, combined with the frequency at which the comparison is made, determines the overhead of applying the EDC, instead of duplication, as a protection against fault attacks.

Various EDCs have been proposed for symmetric and public-key ciphers, most of them being the traditional EDCs described in Chapter 3. In particular, parity-based EDCs were found to be effective for the DES and AES symmetric ciphers. Parity bits can be associated with entire 32-bit words, with individual bytes, or even with nibbles (sets of 4 bits), with each such scheme providing a different fault coverage and entailing a different overhead in terms of extra hardware and delay.

As an example, we illustrate the procedure for developing parity prediction rules when using a parity-based EDC for the AES cipher. Since most data transformations performed in the AES cipher operate on bytes, the natural choice is assigning a parity bit to each byte of the state. This will simplify the prediction rules and provide a high fault coverage. We discuss next the prediction rules for the four steps included in each round.

The prediction of the output parity bits for the ShiftRows transformation is straightforward: it is a rotated version of the input parity bits, following Equation 9.1.

Equally simple is the prediction of the output parity bits of the AddRoundKey step: it consists of adding modulo-2 the input parity matrix associated with the state to the parity matrix associated with the current round key.

The SubBytes step uses SBox lookup tables where each SBox is usually implemented as a 256×8 bits memory. The input to the SBox will already have an associated parity bit. To generate the outgoing parity, a parity bit can be stored with each data byte, increasing the number of bits in each location in the SBox to 9. To make sure that input parity errors are not discarded, we will have to check the parity of the input data and, if an error is detected, stop the encryption process. This would add hardware overhead (parity checkers for 16 bytes) and extra delay.

A better choice would be to propagate the input parity errors so that they can be detected later on. This can be achieved by including the incoming parity bit when addressing the SBox, thus further increasing the table size to 512×9. The entries that correspond to input bytes with correct parity will include the appropriate SubBytes transformation result, with a correct parity bit. The other entries will contain a deliberately incorrect result, such as an all-zeroes byte with an incorrect parity bit.

If fault attacks on the SBox address decoder can be expected, the above scheme is insufficient. In this case, we can add a small and separate table of size 256×1, which will include the predicted parity bit for the correct output byte. This separate table will only allow detection of a mismatch between the parity bit of the correct output byte and the parity bit of the incorrect (but with a valid parity) output byte. We can increase the detection capabilities of this scheme by adding one (or more) correct output data bits to each location in the small table, thus increasing its size. Comparing the output of this table to the appropriate output bits of the main SBox table allows the detection of most addressing circuitry faults.

The output parity bits of the MixColumns step are the most complex to predict. As the reader is requested to verify in the Exercises, the equations for predicting these parity bits are as follows:

$$\begin{aligned}
p_{0,j} &= p_{0,j} \oplus p_{2,j} \oplus p_{3,j} \oplus s_{0,j}^{(7)} \oplus s_{1,j}^{(7)} \\
p_{1,j} &= p_{0,j} \oplus p_{1,j} \oplus p_{3,j} \oplus s_{1,j}^{(7)} \oplus s_{2,j}^{(7)} \\
p_{2,j} &= p_{0,j} \oplus p_{1,j} \oplus p_{2,j} \oplus s_{2,j}^{(7)} \oplus s_{3,j}^{(7)} \\
p_{3,j} &= p_{1,j} \oplus p_{2,j} \oplus p_{3,j} \oplus s_{3,j}^{(7)} \oplus s_{0,j}^{(7)}
\end{aligned} \tag{9.3}$$

where $p_{i,j}$ is the parity bit associated with state byte $s_{i,j}$, and $s_{i,j}^{(7)}$ is the most significant bit of $s_{i,j}$.

The question that remains is the granularity at which the comparisons between the generated and predicted parity bits will be made. Scheduling one validation check at the end of the whole encryption process has the obvious advantage of having the lowest overhead in terms of hardware and extra delay. Theoretically, this could result in the error indication being masked during the encryption procedure, yielding a match between the generated and predicted parity bits despite the ciphertext being erroneous. It can be shown, however, that errors injected at any step of the AES encryption procedure will not be masked, and therefore, a single validation check of the final ciphertext is sufficient for error-detection purposes.

Still, not every combination of errors can be detected by this scheme. Parity-based EDCs are capable of detecting any fault that consists of an odd number of bit errors; an even number of bit errors occurring in a single byte will not be detected. Moreover, if errors are injected in both the state and the round key, some data faults of odd cardinality will not be detected, for example, a single bit error in the round key and a single bit error in the state, occurring in matching bytes which are added in the AddRoundKeys step. The reason we do not restrict our discussion to single bit error coverage (as is usually done when benign faults are considered) is that when a malicious fault injection attack takes place, it most likely impacts multiple adjacent bits of the state and/or round key. Still, although we cannot expect a 100% fault coverage when using a parity-based EDC, the fault coverage has been shown to be very high, even when multiple faults are considered.

Parity-based EDCs are suitable for the DES cipher as well, but the situation here is different from that with AES, due to two of the internal operations in the DES encryption process, namely, the expansion (from 32 to 48 bits) and the permutation of the 32 bits. The latter permutation is irregular, and therefore, there is no simple way to predict the individual parity bits of the four bytes. A more practical solution is to verify the correctness of the permutation by duplicating the circuit and comparing the results. In addition, if we wish to detect faults in the remaining steps of the encryption using a parity-based EDC, we must schedule a validation checkpoint within each round prior to the permutation and generate new parity bits afterward. A simple way to overcome the complexity of parity prediction for the 32-bit permutation is to use a single parity bit per 32-bit word. This, however, yields a very low fault coverage and is not recommended.

In a similar way, EDCs can be developed for other symmetric key ciphers. Several such ciphers that rely on modular addition and multiplication will better match residue codes (see Chapter 3). Other symmetric ciphers have been shown to require a very expensive implementation of EDCs, leading to the conclusion that the brute-force duplication is probably a more suitable solution. The cost of providing protection against fault-based attacks should be taken into account when selecting a cipher for a device.

The RSA public key cipher is based on modular arithmetic operations, and as such, it suggests the residue code as a natural choice. First, the check bits for the plaintext are generated based on the selected modulus C for the residue check

```
Decryption_Algorithm_1($S, N, (d_{n-1}, d_{n-2}, \ldots, d_0)$)
begin
    $a = S$
    for $i$ from $n-2$ to 0 do
        $a = a^2 \bmod N$
        if $d_i = 1$ then $a = S \cdot a \bmod N$
    end
    $M = a$
end
```

FIGURE 9.8 A straightforward decryption algorithm for RSA.

($M \bmod C$ where M is the original message). Since all operations performed during the RSA encryption (and decryption) are modular ones, we can apply them to the input check bits and obtain the predicted output check bits. The residue check will fail to detect an error if the faulty ciphertext has the same residue check bits as the correct one. Assuming that the fault injected is random, this match will happen with a probability of $1/C$, and thus, a higher value of C will result in a higher fault coverage (but also a higher overhead).

9.3.3 Are These Countermeasures Sufficient?

The objective of the countermeasures described above is to detect any fault injected during the process of encryption or decryption, and when such faults are detected, prevent the transmission of the erroneous results that may assist the attacker in extracting the secret key. Unfortunately, it has been demonstrated that although the detection of faults is necessary, it is not always sufficient for protecting against fault-based attacks. We illustrate this point through two examples: an RSA decryption and an AES encryption.

Suppose we use for the RSA decryption a straightforward algorithm that consists of raising the input S to the power d (where d is the private key) as shown in Figure 9.8. The inputs to this algorithm are the encrypted message S, the modulus N, and the n-bit private key $d = d_{n-1}, d_{n-2}, \ldots, d_0$.

■ EXAMPLE

Assume a 4-bit private key $(d_3, d_2, d_1, d_0) = (1011)$ (the decimal 11). The algorithm in Figure 9.8 will calculate $M = ((S^2)^2 \cdot S)^2 \cdot S = S^{11}$. ■

Fault attacks on this algorithm can be detected either by using a residue code or by calculating $M^e \bmod N$ and comparing the result to S. Even with either of

```
Decryption_Algorithm_2($S, N, (d_{n-1}, d_{n-2}, \ldots, d_0)$)
begin
    $a = S$
    for $i$ from $n-2$ to 0 do
        $a = a^2 \bmod N$
        $b = S \cdot a \bmod N$
        if $d_i = 1$ then $a = b$ else $a = a$
    end
    if (*no error has been detected*) then $M = a$
end
```

FIGURE 9.9 A modified decryption algorithm for RSA.

these detection techniques, the algorithm is vulnerable to a power analysis-based attack because a step where $d_i = 0$ will consume less power than a step for which $d_i = 1$. To counter such an attack, the algorithm can be modified so that the power consumed in every step will be independent of d_i. The modified algorithm shown in Figure 9.9 will, as expected, incur higher delay and power penalties compared to the original algorithm. The check at the end of the algorithm intends to make the algorithm resistant to fault injections.

However, a careful examination of the algorithm in Figure 9.9 reveals that it is still vulnerable to fault-based attacks. Since the result b of the multiplication $S \cdot a \bmod N$ is not used if $d_i = 0$, the attacker can inject a fault during this multiplication, and if the final result of the decryption is correct, he can deduce one bit of the secret private key.

Fortunately, a different algorithm can be devised using what is called a Montgomery ladder, as shown in Figure 9.10. In this algorithm, the intermediate values of both a and b are used in the next step, and thus, a fault injected in any intermediate step will yield an erroneous result which will be detected.

EXAMPLE

Assume, as before, a 4-bit private key $(d_3, d_2, d_1, d_0) = (1011)$. The algorithm in Figure 9.10 will calculate M as follows. For $i = 3$, $d_3 = 1$, and thus, $a = S$ and $b = S^2$. For $i = 2$, $d_2 = 0$, and thus, $a = S^2$ and $b = S^3$. For $i = 1$, $d_1 = 1$, and thus, $a = S^5$ and $b = S^6$. Finally, for $i = 0$, $d_1 = 1$, resulting in $M = a = S^{11}$ and $b = S^{12}$. ■

The Montgomery-ladder-based decryption algorithm for RSA allows another approach to detect faults injected during the decryption. The computed a and b must be of the form (M, SM), and a fault injected during any intermediate step will destroy this relationship. Thus, checking whether a and b satisfy this relationship before providing the final result of the decryption can detect all injected errors,

```
Decryption_Algorithm_3($S, N, (d_{n-1}, d_{n-2}, \ldots, d_0)$)
begin
    $a = 1$
    $b = S$
    for $i$ from $n-1$ to 0 do
        if $d_i = 0$ then
            $a = a^2 \bmod N$
            $b = a \cdot b \bmod N$
            end
        if $d_i = 1$ then
            $a = a \cdot b \bmod N$
            $b = b^2 \bmod N$
            end
    end
    if (*no error has been detected*) then $M = a$
end
```

FIGURE 9.10 A Montgomery-ladder-based decryption algorithm for RSA.

except those that modify either the bits of the secret private key d or the number of times the loop in Figure 9.10 is performed. By using some EDC for these two, in addition to verifying the relationship between a and b, all injected faults can be detected.

We next describe a fault-based attack on AES encryption that may succeed even if a fault-detection mechanism that prevents erroneous results from being output is incorporated into the design. The attack starts with providing an all-zeroes input to the AES encryption device. In the very first step of the encryption (see Figure 9.4), the initial round key is added, resulting in the state matrix $s_{i,j} = 0 \oplus k_{i,j} = k_{i,j}$, where $0 \leqslant i,j \leqslant 3$. At exactly the same time instant, before the first SubBytes operation, the attacker injects a fault into the ℓth bit ($\ell = 0,1,\ldots,7$) of a particular byte $s_{i,j}$ of the state matrix so that the selected bit is set to 0. If the corresponding bit of the key (bit ℓ of $k_{i,j}$) is 1, the output will be incorrect and the detection mechanism will disallow this output. If, however, the corresponding bit of the key is 0, no error will occur and the encryption device will work properly, providing the attacker with the value of that bit of the secret key.

This attack is very simple to understand theoretically but may prove to be quite difficult to mount due to the need for precise timing and location of the injected fault. The secret key can still be extracted even if the strict timing and location requirements of this attack are relaxed, but this may require a larger number of fault injection experiments. The interested reader can find further details in the original paper referenced in the Further Reading section. The simple attack described above shows that implementations of symmetric key ciphers, even those with fault detection capabilities, are not completely immune to fault-based attacks.

9.3.4 Final Comment

A final remark is in order: the topic of this chapter is still a very active area of research and a constant stream of new fault-based attacks on cryptographic devices, and of novel countermeasures to protect the devices against these attacks appears in the literature. The objective of this chapter is to demonstrate the extra difficulties in devising fault-protection techniques to deal with malicious faults injected into cryptographic devices.

9.4 Further Reading

The official descriptions of the DES and AES algorithms appear in [24] and [25], respectively. The AES example that is outlined in Section 9.1.1 is detailed in [25]. A more detailed description of AES appears in [13]. The RSA algorithm was first described in [27]. Javascript AES, DES, and RSA calculators/demonstrators showing intermediate values are available [29]. A considerable number of articles on all aspects of cryptography are posted on the Website of the International Association for Cryptologic Research [17]. Well-written descriptions of key terms in cryptography appear in the online encyclopedia *Wikipedia* [30].

Fault injection attacks were first discussed in [7]. Many other fault attacks on public and symmetric key ciphers have been later presented in [1,2,9,12,14, 16,26,32]. A survey of various fault injection techniques is provided in [3] which also reviews some protection schemes against such attacks. Detailed descriptions of ways to protect ciphers from attacks appear in [4,5,8,11,19–21,23,28]. The derivation of the parity bit prediction rules for AES follows [4]. Simulators for error detection in several ciphers are available online [22]. The insufficiency of fault detection schemes against fault-based attacks on RSA and AES has been demonstrated in [8,31]. The modified RSA decryption algorithm based on the Montgomery ladder is described in [15,18]. New fault injection attacks and countermeasures appear in [10].

9.5 Exercises

1. Construct an RSA encryption scheme using $p = 61$ and $q = 53$. Select the public key $e = 17$, which is obviously relatively prime to $\phi(pq)$. Find the corresponding private key d, and for the message $M = 123$, calculate the ciphertext and show that the private key allows the decryption of the ciphertext.

2. Develop a software implementation of DES (or find one on the Internet) and apply the fault-based attack shown in Table 9-2. Modify the program to inject the faults, and write another program to find the secret key.

3. Complete the example (in the chapter) of injecting a fault into the private key d of an RSA decryption device that uses the public key $(e,N) = (7,77)$ and

the private key $(d,N) = (43,77)$. Assume a ciphertext of 37 as in the example. List all possible single-bit errors and all double-bit errors that can be injected into d. For each error on your list find the erroneous plaintext that the device will produce. Are all the erroneous plaintexts unique?

4. Develop a software implementation of RSA (or find one on the Internet), use the prime numbers $p = 7$ and $q = 11$ as in the example in this chapter and select $e = 7$. This yields the public key $(e,n) = (7,77)$ and the private key $(d,n) = (43,77)$. Inject single-bit failures in your program, and obtain all the bits of the private key.

5. Use the program and parameters from Problem 4 and add a residue check with the modulus 3. Repeat the single-bit fault attacks. Will the modified program detect all such faults?

6. Show that $x^8 \bmod g(x) = x^4 + x^3 + x + 1$ for the generator polynomial of AES $g(x) = x^8 + x^4 + x^3 + x + 1$.

7. Verify all 16 results of the MixColumns step that are shown in Figure 9.6f.

8. Inject a single-bit error in the state matrix shown in Figure 9.6c, replacing the first byte 19 by 18, and calculate the erroneous state matrix at the end of round 2. Compare your result to the matrix shown in Figure 9.6h. How many bytes are in error?

9. Suppose you are using AES with data blocks and key of size 128 bits. Your messages however are only 50-bit long. What would you put in the unused 78-bit positions?

10. Verify the correctness of the parity prediction equations for the MixColumns step in AES.

References

[1] R. Anderson and M. Kuhn, “Low Cost Attacks on Tamper Resistant Devices,” *International Workshop on Security Protocols*, Lecture Notes in Computer Science, Vol. 1361, pp. 125–136, Springer-Verlag, 1997.

[2] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert, “Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures,” *Cryptology ePrint Archive,* Report 2002/073, 2002. Available at: http://eprint.iacr.org/2002/073.

[3] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, “The Sorcerer’s Apprentice Guide to Fault Attacks,” *Proceedings of the IEEE,* Vol. 94, Issue 2, pp. 370–382, February 2006. Also in the *Cryptology ePrint Archive,* Report 2004/100, 2004. Available at: http://eprint.iacr.org/2004/100.

[4] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, and V. Piuri, “Error Analysis and Detection Procedures for a Hardware Implementation of the Advanced Encryption Standard,” *IEEE Transactions on Computers*, Vol. 52, pp. 492–505, April 2003.

[5] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, and V. Piuri, "Concurrent Fault Detection in a Hardware Implementation of the *RC*5 Encryption Algorithm," *IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pp. 410–419, 2003.

[6] G. Bertoni, L. Breveglieri, I. Koren, and P. Maistri, "An Efficient Hardware-Based Fault Diagnosis Scheme for AES: Performances and Cost," *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pp. 130–138, October 2004.

[7] E. Biham and A. Shamir, "Differential Fault Analysis of Secret Key Cryptosystems," *17th Cryptology Conference, Crypto 97, Lecture Notes in Computer Science*, Vol. 1294, pp. 513–525, Springer-Verlag, 1997.

[8] J. Blöemer and J.-P. Seifert, "Fault Based Cryptanalysis of the Advanced Encryption Standard (AES)," *Financial Cryptography, Lecture Notes in Computer Science*, Vol. 2742, pp. 162–181, Springer-Verlag, 2003. Available at: http://eprint.iacr.org/2002/075.

[9] D. Boneh, R. DeMillo, and R. Lipton, "On the Importance of Eliminating Errors in Cryptographic Computations," *Journal of Cryptology*, Vol. 14, pp. 101–119, 2001.

[10] L. Breveglieri, I. Koren, D. Naccache, and J.-P. Seifert (Eds.), *Fault Diagnosis and Tolerance in Cryptography (FDTC), Lecture Notes in Computer Science*, Vol. 4236, Springer-Verlag, 2006.

[11] A. S. Butter, C. Y. Kao, and J. P. Kuruts, "DES Encryption and Decryption Unit with Error Checking," US patent US5432848, July 1995.

[12] M. Ciet and M. Joye, "Elliptic Curve Cryptosystems in the Presence of Permanent and Transient Faults," *Cryptology ePrint Archive,* Report 2003/028, 2003. Available at: http://eprint.iacr.org/2003/028.

[13] J. Daemen and V. Rijmen, *The Design of Rijndael: AES—The Advanced Encryption Standard,* Springer-Verlag, 2002.

[14] C. Giraud, "DFA on AES," *Cryptology ePrint Archive,* Report 2003/008, 2003. Available at: http://eprint.iacr.org/2003/008.

[15] C. Giraud, "Fault Resistant RSA Implementation," *Fault Diagnosis and Tolerance in Cryptography (FDTC'05),* pp. 143–151, 2005.

[16] C. Giraud and H. Thiebeauld, "Basics of Fault Attacks," *Fault Diagnosis and Tolerance in Cryptography (FDTC'04)—Supplemental Volume of the Dependable Systems and Networks Conference,* pp. 343–347, 2004.

[17] International Association for Cryptologic Research. Available at: http://www.iacr.org/. ePrint Archive, available at: http://eprint.iacr.org.

[18] M. Joye and S.-M. Yen, "The Montgomery Powering Ladder," Cryptographic Hardware and Embedded Systems—CHES 2002, *Lecture Notes in Computer Science*, Vol. 2523, pp. 291–302, Springer-Verlag, 2002.

[19] R. Karri, K. Wu, P. Mishra, and K. Yongkook, "Fault-Based Side-Channel Cryptanalysis Tolerant Rijndael Symmetric Block Cipher Architecture," *IEEE Symposium on Defect and Fault Tolerance in VLSI Systems*, pp. 427–435, 2001.

[20] R. Karri, G. Kuznetsov, and M. Goessel, "Parity-based Concurrent Error Detection in Symmetric Block Ciphers," *International Test Conference 2003—ITC 2003*, Vol. 1, ISSN 1089-3539, pp. 919–926, 2003.

[21] M. G. Karpovsky and A. Taubin, "A New Class of Nonlinear Systematic Error Detecting Codes," *IEEE Transactions on Information Theory,* Vol. 50, pp. 1818–1820, 2004.

[22] I. Koren, Fault Tolerant Computing Simulator. Available at: http://www.ecs.umass.edu/ece/koren/fault-tolerance/simulator/.

[23] K. J. Kulikowski, M. G. Karpovsky, and A. Taubin, "Fault Attack Resistant Cryptographic Hardware with Error Detection," *Fault Diagnosis and Tolerance in Cryptography (FDTC'06), Lecture Notes in Computer Science*, Vol. 4236, pp. 185–195, Springer-Verlag, 2006.

[24] National Institute of Standards and Technology, "Data Encryption Standard," *FIPS Publication No. 46*, January, 1977.

[25] National Institute of Standards and Technology, "Advanced Encryption Standard," *FIPS publication No. 197*, November 2001. Available at: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[26] G. Piret and J.-J. Quisquater, "A Differential Fault Attack Technique against SPN Structures, with Application to the AES and Khazad," Cryptographic Hardware and Embedded Systems—CHES 2003, *Lecture Notes in Computer Science*, Vol. 2779, pp. 77–88, Springer-Verlag, 2003.

[27] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-key Cryptosystems," *Communications of the ACM*, Vol. 21, pp. 120–126, 1978.

[28] A. Shamir, "Method and Apparatus for Protecting Public Key Schemes from Timing and Fault Attacks," US Patent 5991415, 1999.

[29] E. Styer, AES calculator, available at: http://www.cs.eku.edu/faculty/styer/460/Encrypt/JS-AES.html; DES calculator, available at: http://www.cs.eku.edu/faculty/styer/460/Encrypt/JS-DES.html; RSA demonstrator, available at: http://wwwr.cs.eku.edu/faculty/styer/460/Encrypt/RSAdemo.html.

[30] *Wikipedia, The Free Encyclopedia*. Available at: http://en.wikipedia.org/wiki/Cryptography.

[31] S.-M. Yen and M. Joye, "Checking Before Output May Not Be Enough Against Fault-Based Cryptanalysis," *IEEE Transactions on Computers*, Vol. 49, pp. 967–970, September 2000.

[32] S.-M. Yen, S. Moon, and J.-C. Ha, "Permanent Fault Attack on the Parameters of RSA with CRT," *Lecture Notes in Computer Science,* Vol. 2727, pp. 285–296, Springer-Verlag, 2003.