The CSI Journal on
Computer Science and Engineering
Vol. 4, No. 2&4, 2006
Pages 32- 38
Regular Paper

From Fault Tolerance to Fault Attack Tolerance in the Implementations of Advanced Encryption Standard

Amir Moradi[1], Mahmoud Salmasizadeh[2], Mohammad Taghi Manzuri-Shalmani[1, 3]

[1] Department of Computer Engineering, Sharif University of Technology, Tehran, Iran
[2] Electronic Research Center, Sharif University of Technology, Tehran, Iran
[3] School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran

Abstract

Fault attack techniques are powerful and efficient cryptanalysis methods to find the secret key of cryptographic devices. Thus, several methods have been introduced to offset this type of side channel attack. On the other hand, some techniques were presented to locate and detect faults in the implementations of symmetric and asymmetric encryption/decryption algorithms. To our best knowledge, this paper is the first article which examines the effectiveness of fault tolerance techniques to prevent fault attacks. Also, we introduce a minimum time redundant method of using the inverse modules for Concurrent Error Detection (CED). The usage of Error Correction Codes (ECC) in implementations of Advanced Encryption Standard (AES) is another approach that is proposed in this article. We present the comparison between the usage of the proposed ECCs to make fault tolerant implementation and to resist it against fault attacks. Experimental results of one of the proposed ECCs show that almost all possible faults are detected, while some of them are corrected. Thus, it resists against approximately all injected faults to attack on the implementation of AES algorithm.

Keywords: ECC, Fault Tolerance, Fault Masking, Side-Channel Attack Countermeasure, DFA, AES, CED.

1. Introduction

Nowadays, side channel attacks can find the secret key of the implementation of many cryptosystems especially in dedicated cryptographic hardware modules such as smart cards. Fault attack is one of the side channel attacks that uses the faulty ciphertexts caused by malicious injected faults. In this type of attack, the attacker compares the faulty ciphertexts with fault free ciphertext and uses their difference to discover some intermediate secret values or the main key. Consequently, it is called Differential Fault Attack (DFA) [1]. The Rijndael Advanced Encryption Standard (AES) [10] is the most popular and widely used symmetric cryptosystem applied in standard communication devices such as Ethernet switches, routers, smart cards and etc. This algorithm has been taken into consideration by the attackers and researchers to verify its security and to find its implementation weaknesses. Therefore, many methods have been presented to break the AES implementations by DFA [2, 3, 4, 5]. Then, activities to resist fault attacks have been increased and some techniques have been introduced to detect and locate faults in implementations of AES [6, 7, 8, 9]. Some of the presented methods use time redundancy to compare and detect the fault occurrence. The inverse modules which are used at each unit of encryption/decryption algorithm can locate faults by comparing the input of each unit with the result of the inverse unit. Figure 1 shows the comparison schema for one round of Rijndael encryption algorithm [7].

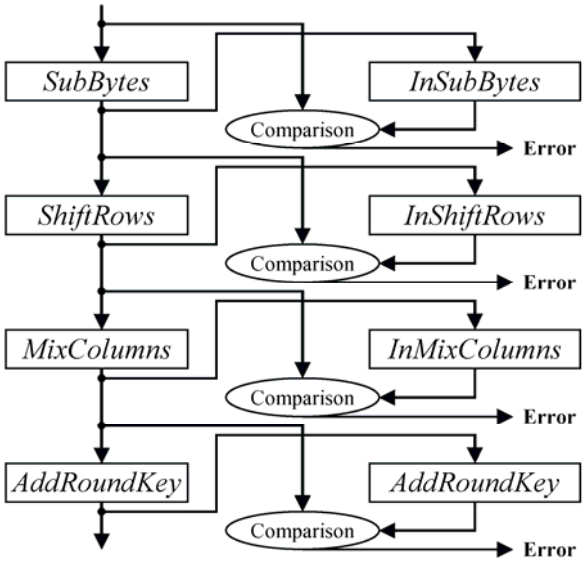

Figure 1. Operational level CED in one round of the AES

Other methods have applied error detection parity codes to locate and detect odd order faults. Parity check codes help fault detection techniques to avoid faulty ciphertexts. Then, the attacker can not employ DFA without faulty ciphertexts. The main aim of this article is to compare the fault tolerance techniques and the fault attack tolerance methods. We show that fault tolerance techniques are not effective to counter differential fault attacks. Also, we propose a technique to use inverse modules with the least time latency. The latency of the proposed method is negligible in pipeline architectures. On the other hand, we present three ECCs to be used in the implantation of AES encryption algorithm; then, comparison between the proposed methods and the previous techniques is illustrated. We show that the previous fault tolerance methods are not effective to counter the attackers who inject the faults to discover the secret key. Additionally, we compare the effects of the time redundant methods with the other techniques which use error detection/correction codes (including our proposed ones) to counter fault attacks.

This paper is organized as follows: In section 2, the proposed concurrent error detection method using inverse modules with minimum time overhead is presented. In section 3, a brief overview of error correction codes are described, and three ECCs are proposed to use in the AES implementations. Section 4 illustrates the difference between fault tolerance and fault attack tolerance methods that apply various fault detection and correction schemes. The required relations and equations of one of the proposed ECCs for each function and unit of the AES encryption algorithm are explained in section 5. The experimental results of using the proposed ECC, its effect on fault (attack) tolerance, and its area overhead are presented in section 6. Finally, section 7 concludes the paper.

2. Low Latency CED Scheme

According to the Figure 1, the inverse module of each function (*SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey*) computes the inverse of the function; also, the next operation will not start until the input of the function and the output of the inverse function pass true from the comparison. Thus, the usage of this method leads to a big latency approximately equal to the original one. Then, the needed time to encrypt/decrypt will be doubled exactly. Figure 2 shows our method that uses inverse modules. In this case, inverse modules operate one step later. Consequently, faults are detected such as the previous method but with only one unit latency. However, the whole latency depends on the implementation architecture. The total latency of the algorithm equals to the biggest latency of individual functions of the AES cryptosystem.

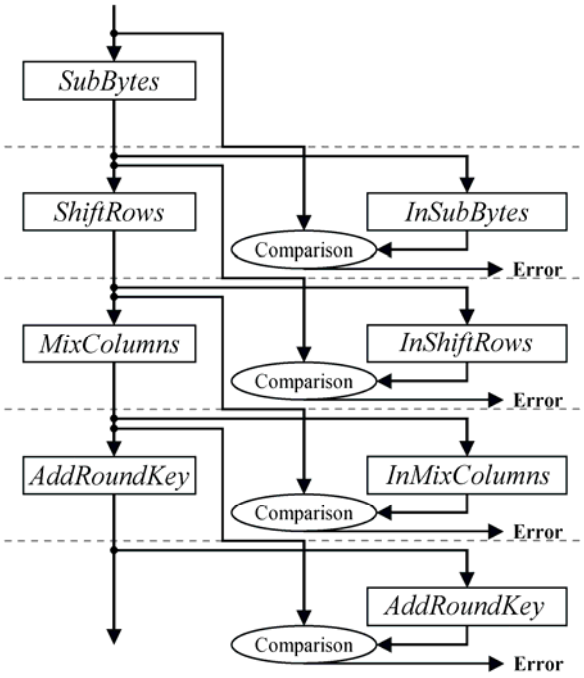

Figure 2. Low-latency operational level CED in one round of the AES encryption algorithm

The proposed method was simulated using Verilog® HDL on a Xilinx Virtex2 chip. In this implementation, the latency that is added to whole of the algorithm was one time slot. Also, the proposed method is efficient in pipeline architectures. Cryptographic devices have often both encryption and decryption modules; these methods (using the inverse modules) are cost effective if the inverse sub modules exist in the cryptographic unit. Table 1 shows the area and time overhead of applying this technique on two different architectures in comparison with the previous methods. In the presented results we assumed that the architectures have the inverse modules and functions.

Table 1. Area and time overheads of applying the proposed method on the two AES implementations

Type	Area Overhead %			Time Overhead
Architecture	Slice	Flip Flop	LUT	Clock Pulse
1	38	0	33	1
2	29	23	31	4

3. ECCs and Reed-Solomon Codes

In 1960, I. Reed and G. Solomon developed the **block code** coding technique called Reed-Solomon (RS) coding. Today, RS codes remain popular due to standards compliance and economic implementations [11, 12]. In the construction of the RS code, we assume that messages are represented as sequences of length n of numbers in $\mathbf{Z}_q$, where q is a prime. We will encode each of these n sequences into a code of length m. With each sequence

$$a = (a_{n-1}, a_{n-2}, ..., a_1, a_0) \in Z_q^n$$

We associate a degree n-1 polynomial f_a defined as follows:

$$f_a(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + ... + a_1x + a_0 \quad (1)$$

The message sequence of values of f_a at m distinct points in $\mathbf{Z}_q$:

$$\hat{a} = (f_a(m), f_a(m-1), ..., f_a(2), f_a(1)) \quad (2)$$

Thus, $q \geq m$. The number of correctable symbols, *i.e. t*, is given by equation (3).

$$t = \left\lfloor \frac{m}{2} \right\rfloor \quad (3)$$

For example, consider messages of length 2 over $\mathbf{Z}_5$, *i.e.*, n=2 and q=5. Suppose that the encoded message is a sequence of length m=3. Table 2 shows the polynomials and codewords for three examples. We selected two types of RS codes and one Hamming Code to use in the AES functions. Each proposal has different coverage rate in detection and correction. In the subsections we introduce each of them, and we compare their capabilities.

Table 2. Three examples for RS code, n=2, q=5

Message	Polynomial	Codeword
(1, 4)	x+4	(2, 1, 0)
(4, 3)	4x+3	(0, 1, 2)
(4, 2)	4x+2	(4, 0, 1)

3.1. The First Proposed Error Correction Code (RS1)

In this model, we add 8-bit redundancy to each 8 bits of information. In below equation I is the data information and R is the added redundancy. I consists of two 4-bit parts: I_2 (most significant nibble) and I_1 (least significant nibble). Also, R includes two nibbles: R_2 and R_1.

$$R_1 = I_2 \oplus I_1 \quad , \quad R_2 = 2 \cdot I_2 \oplus I_1 \quad (4)$$

Suppose that it is used in a data communication. The receiver calculates R' on the base of equation (4) after it gets I and R. If R equals R', no error has occurred probably. Otherwise, the receiver must ignore this packet or try to correct it. In below equations, E is the occurred error that must be added to I to correct the information received (all calculations in equations below are in 4-bit).

$$E_1' = E_1 \oplus 2 \cdot E_1 = I_1 \oplus 2 \cdot I_1 \oplus 2 \cdot R_1 \oplus R_2 \quad (5)$$

$$E_2' = E_2 \oplus 2 \cdot E_2 = I_2 \oplus 2 \cdot I_2 \oplus R_1 \oplus R_2 \quad (6)$$

$$E_i = E_i' \oplus 2 \cdot E_i' \oplus 4 \cdot E_i' \oplus 8 \cdot E_i' \quad (7)$$

$$E = 0 \quad \Rightarrow \quad R = R' \quad (8)$$

According to the equation (3), this redundancy code can correct 4-bit errors. That is, when one of E_1 and E_2 is zero, the occurred error is correctable. We define two modules called ***Check_Redundancy*** and ***Correct_Redundancy***. The ***Check_Redundancy*** module can be used when we need to detect uncorrectable faults, but ***Correct_Redundancy*** module tries to correct errors without considering uncorrectable ones. Seems that ***Correct_Redundancy*** is not suitable for communication; we will illustrate its usage in the next section.

3.2. The Second Used ECC (RS2)

In this case, 6-bit redundancy is added to each 8-bit data. I consists of three parts: I_3(2 most significant bits), I_2(3-bit middle part), and I_1(3 least significant bits). Also, R includes two 3-bit parts: R_2 and R_1.

$$R_1 = I_3 \oplus I_2 \oplus I_1 \quad , \quad R_2 = 4 \cdot I_3 \oplus 2 \cdot I_2 \oplus I_1 \quad (9)$$

Same as RS1, the receiver should calculate E. However, this calculation is not as easy as RS1 (all calculations in equations below are in 3-bit).

$$E_3 = 0 \Rightarrow \begin{vmatrix} E_1' = E_1 \oplus 2 \cdot E_1 \\ E_2' = E_2 \oplus 2 \cdot E_2 \end{vmatrix} \Rightarrow$$

$$\begin{vmatrix} E_1' = I_1 \oplus 2 \cdot I_1 \oplus 2 \cdot I_3 \oplus 4 \cdot I_3 \oplus 2 \cdot R_1 \oplus R_2 \\ E_2' = I_2 \oplus 2 \cdot I_2 \oplus I_3 \oplus 4 \cdot I_3 \oplus R_1 \oplus R_2 \\ E_i = E_i' \oplus 2 \cdot E_i' \oplus 4 \cdot E_i' \end{vmatrix} \quad (10)$$

$$E_2 = 0 \Rightarrow \begin{vmatrix} E_1' = E_1 \oplus 4 \cdot E_1 \\ E_3' = E_3 \oplus 4 \cdot E_3 \end{vmatrix} \Rightarrow$$

$$\begin{vmatrix} E_1' = I_1 \oplus 4 \cdot I_1 \oplus 2 \cdot I_2 \oplus 4 \cdot I_2 \oplus 4 \cdot R_1 \oplus R_2 \\ E_3' = I_2 \oplus 2 \cdot I_2 \oplus I_3 \oplus 4 \cdot I_3 \oplus R_1 \oplus R_2 \\ E_i = E_i' \oplus 4 \cdot E_i' \end{vmatrix} \quad (11)$$

$$E_1 = 0 \Rightarrow \begin{vmatrix} E_2' = 2 \cdot E_2 \oplus 4 \cdot E_2 \\ E_3' = 2 \cdot E_3 \oplus 4 \cdot E_3 \end{vmatrix} \Rightarrow$$

$$\begin{vmatrix} E_2' = I_1 \oplus 4 \cdot I_1 \oplus 2 \cdot I_2 \oplus 4 \cdot I_2 \oplus 4 \cdot R_1 \oplus R_2 \\ E_3' = I_1 \oplus 2 \cdot I_1 \oplus 2 \cdot I_3 \oplus 4 \cdot I_3 \oplus 2 \cdot R_1 \oplus R_2 \\ 2 \cdot E_i = E_i' \oplus 2 \cdot E_i' \end{vmatrix} \quad (12)$$

According to equation (3), RS2 can correct 3-bit errors. The happened error is correctable when two of E_1 to E_3 are zero. In other words, only one of E_1 to E_3 must be non-zero.

3.3. The Third Code (HC)

In the last model, we apply a 4-bit redundancy of the Hamming Code for each byte. The number of correctable and detectable errors in Hamming Codes is given by (13). *C*, *D* and *HD* are the number of correctable, detectable and hamming distance respectively.

$$C + D + 1 = HD \tag{13}$$

Thus, the added redundancy can detect 3 bits and correct 1 bit errors. We ignored essential HC equations and proof of RS1 and RS2 for simplicity.

We implemented and simulated all three proposed ECCs for all possible values of information and error. Table 3 shows the coverage rate of each ECC for the ***Check_Redundancy*** module. In this table, the **UnCorrectable** column shows the number of errors that are detectable but are not correctable. The **BadAction** means that the module tried to correct but the modified data was not. The **PassTrue** is related to the situation that, an error happened on the redundancy part then module detects it and passes the information without any change. We should describe that the HC was considered as an error detection scheme only in the ***Check_Redundancy*** and it can not correct and detect errors simultaneously. Additionally, last two columns of the table 3 show the needed area for FPGA implementation.

4. Using ECCs for Fault (Attack) Tolerance

In this section we discuss difference between fault tolerance and making fault attack tolerant implementations. When the attacker wants to use faulty ciphertext, tries to inject faults. In this case, the tolerant implementation must prevent the production of useful faulty ciphertexts. However, fault tolerant systems can detect or correct some occurred faults with environmental reasons not malicious. For example the presented method to use parity check bits in [6, 8] can detect all odd order faults in each byte. This technique will detect many faults and prevents to generate faulty ciphertext. But we assert that it is not impressive to counter fault attacks. According to the presented attack in [5, 13], the attacker needs faulty ciphertexts that caused by faults on the first byte of *MixColumns* input in the 9th round. She injects faults to this location, so odd order faults on this byte are detectable and she will not obtain faulty ciphertexts. But in other faults, faulty ciphertexts will be created and will be used by the attacker. Then the halves of all possible faults in each byte are useful to attack. Thus, the attacker can try to inject faults until she reaches the faulty ciphertext. Other methods that use inverse modules to detect faults such as the proposed technique in section 2 are in different situation. In these cases, approximately all faults will be detected and only 0.39% of all possible faults in a byte are not detectable. In this situation, the attacker will not be able to gain faulty ciphertext very likely. If the inverse modules do not exist in the architecture, these methods are not profitable and adding them leads to double required resources and area.

Applying ECCs helps to detect many faults and correct some of them. We propose to use the ***Check_Redandancy*** module of the proposed ECCs to make fault tolerant systems. According to the table 3, just 0.39%, 1.56% and 6.13% of all possible faults are not detectable for the proposed ECCs respectively. Disadvantage of them is the **BadAction** ratio; it will be decreased by modifying the ***Check_Redandancy*** modules. But this modification will increase the essential area. The proposed HC has the minimum area overhead and we offer it to use for fault tolerant implementation of the AES. Table 4 shows the simulation results of the ***Correct_Redundancy*** modules. The results show that the occurred faults lead to **BadAction** very likely. Means, a little number of faults are correctable, but almost all possible faults cause to generate other faulty outputs. This idea is useful to resist against fault attacks. Often, the attacker uses specified fault model for attacking. For example, the used fault model in [4, 2] is only one-bit errors in each byte of *MixColumns* input. With the proposed ***Correct_Redundancy*** modules we will correct all one bit and many other faults. Additionally, the uncorrectable faults will be changed to the other faults and the modified output of each byte will not probably be included by desired fault model. We offer ***Correct_Redundancy*** modules for fault attack tolerant implementation. Comparison between tables 3 and 4 shows the needed area for ***Correct_Redundancy*** modules is less than other ones. Applying these modules are effective in fault tolerant implementations because environmental occurred faults are almost in low bit count and these modules can correct many faults. As described previously, the RS1, RS2 and HC can correct all 4-bit, 3-bit and one-bit faults respectively.

5. Using RS1 on the AES

To employ the proposed ECCs in the AES implementation, the requisite equations for the AES functions must be designed. We selected RS1 for implementation because its redundancy is same as the AES definitions (8 bits) and this ECC generates 8-bit unique redundancy for each 8-bit data. In continue, we present the essential equations for *SubBytes*, *ShiftRows*, *MixColumns* and *AddRoundKey* operations. For more information about the AES details see [10].

5.1. SubBytes

In many architectures, **S-box** functions were implemented with lookup table for its high nonlinearity. These architectures were almost designed for FPGA implementations. Thus, we created another table for redundancy. Means, redundancy of input data is given to the **SR-box** table and the output data redundancy will be generated. Then, the ***Check_Redundancy*** or the ***Correct_Redundancy*** modules can examine the received data and redundancy for detection or correction. Table 5 presents the lookup table for **SR-box.**

Table 3. The coverage rate and the essential area for the used ECCs in the *Check_Redundancy*

Code	Correctable	UnCorrectable	BadAction	PassTrue	UnDetectable	Slice	**LUT**
RS1	0.05%	76.56%	22.96%	0.04%	0.39%	20	**34**
RS2	0.10%	53.14%	45.13%	0.07%	1.56%	9	**16**
HC	0	93.75%	0	0.13%	6.12%	6	**11**

Table 4. The coverage rate and the needed area of *Correct_Redundancy* modules

Code	Correctable	BadAction	UnDetectable	Slice	**LUT**
RS1	0.39%	99.22%	0.39%	6	**11**
RS2	0.38%	98.06%	1.56%	7	**11**
HC	0.48%	93.40%	6.12%	12	**21**

Table 5. The SR-box lookup table

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

5.2. ShiftRows

Because the *ShiftRows* operation changes the location of the bytes in each row, we should displace the redundancy parts according to the movement of the related data. In other words, we must add another *ShiftRows* module for the redundancy part. Then, the correction or detection modules can be employed to process the shifted data and redundancies.

5.3. AddRoundKey

We describe the *AddRoundKey*'s equations before the *MixColumns* because we will use its results in the next subsection. The *AddRoundKey* runs the Exclusive-OR operation on two operands. In consequence, the redundancy of two operands must be added in module 2 same as *AddRoundKey* to create the redundancy of the output data, because *AddRoundKey* is a linear operation.

5.4. MixColumns

Equation (14) illustrates the operation of the *MixColumns* in one column. It uses the polynomial multiplication ($\bullet$) by 2 and 3. The results of the multiplications will be added in module 2. Thus, the redundancy of the added parts will be computable such as *AddRoundKey* operation. Consequently, we should define relations between the redundancy of data and the redundancy of multiplied data by 2 and 3. Equations (15) to (19) introduce the technique to calculate the redundancy of $I \bullet 2$ and $I \bullet 3$. I_i ($7 \geq i \geq 0$) denotes the ith bit of the 8-bit data information, (I). R_i ($7 \geq i \geq 0$) is the ith bit of the 8-bit redundancy of data information, *i.e. R*. Also, $R(I \bullet 2)$ and $R(I \bullet 3)$ represent the redundancy of $I \bullet 2$ and $I \bullet 3$ respectively. Bits of $R(I \bullet 2)$ and $R(I \bullet 3)$ are represented by $RI2_i$ and $RI3_i$.

$$M\left(\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix}\right) = \begin{bmatrix} 2 \bullet a_1 \oplus 3 \bullet a_2 \oplus a_3 \oplus a_4 \\ a_1 \oplus 2 \bullet a_2 \oplus 3 \bullet a_3 \oplus a_4 \\ a_1 \oplus a_2 \oplus 2 \bullet a_3 \oplus 3 \bullet a_4 \\ 3 \bullet a_1 \oplus a_2 \oplus a_3 \oplus 2 \bullet a_4 \end{bmatrix} \tag{14}$$

$$\begin{vmatrix} R_0 = I_0 \oplus I_4 \\ R_1 = I_1 \oplus I_5 \\ R_2 = I_2 \oplus I_6 \\ R_3 = I_3 \oplus I_7 \end{vmatrix}, \begin{vmatrix} R_4 = I_0 \\ R_5 = I_1 \oplus I_4 \\ R_6 = I_2 \oplus I_5 \\ R_7 = I_3 \oplus I_6 \end{vmatrix} \tag{15}$$

$$\begin{vmatrix} (I \bullet 2)_0 = I_7 \\ (I \bullet 2)_1 = I_0 \oplus I_7 \\ (I \bullet 2)_2 = I_1 \\ (I \bullet 2)_3 = I_2 \oplus I_7 \end{vmatrix}, \begin{vmatrix} (I \bullet 2)_4 = I_3 \oplus I_7 \\ (I \bullet 2)_5 = I_4 \\ (I \bullet 2)_6 = I_5 \\ (I \bullet 2)_7 = I_6 \end{vmatrix} \tag{16}$$

$$\begin{aligned}
&RI2_0 = I_3 = R_0 \oplus R_1 \oplus R_2 \oplus R_4 \oplus R_5 \oplus R_6 \oplus R_7 \\
&RI2_1 = R_0 \oplus I_7 = R_1 \oplus R_2 \oplus R_3 \oplus R_4 \oplus R_5 \oplus R_6 \oplus R_7 \\
&RI2_2 = R_1 \\
&RI2_3 = R_2 \oplus I_7 = R_0 \oplus R_1 \oplus R_3 \oplus R_4 \oplus R_5 \oplus R_6 \oplus R_7 \\
&RI2_4 = I_7 = R_0 \oplus R_1 \oplus R_2 \oplus R_3 \oplus R_4 \oplus R_5 \oplus R_6 \oplus R_7 \\
&RI2_5 = R_4 \oplus I_3 = R_0 \oplus R_1 \oplus R_2 \oplus R_5 \oplus R_6 \oplus R_7 \\
&RI2_6 = R_5 \\
&RI2_7 = R_6 \oplus I_7 = R_0 \oplus R_1 \oplus R_2 \oplus R_3 \oplus R_4 \oplus R_5 \oplus R_7
\end{aligned} \tag{17}$$

$$R(I \bullet 3) = R \oplus R(I \bullet 2) \tag{18}$$

$$\begin{aligned}
&RI3_0 = RI2_0 \oplus R_0 = R_1 \oplus R_2 \oplus R_4 \oplus R_5 \oplus R_6 \oplus R_7 \\
&RI3_1 = RI2_1 \oplus R_1 = R_2 \oplus R_3 \oplus R_4 \oplus R_5 \oplus R_6 \oplus R_7 \\
&RI3_2 = RI2_2 \oplus R_2 = R_1 \oplus R_2 \\
&RI3_3 = RI2_3 \oplus R_3 = R_0 \oplus R_1 \oplus R_4 \oplus R_5 \oplus R_6 \oplus R_7 \\
&RI3_4 = RI2_4 \oplus R_4 = R_0 \oplus R_1 \oplus R_2 \oplus R_3 \oplus R_5 \oplus R_6 \oplus R_7 \\
&RI3_5 = RI2_5 \oplus R_5 = R_0 \oplus R_1 \oplus R_2 \oplus R_6 \oplus R_7 \\
&RI3_6 = RI2_6 \oplus R_6 = R_5 \oplus R_6 \\
&RI3_7 = RI2_7 \oplus R_7 = R_0 \oplus R_1 \oplus R_2 \oplus R_3 \oplus R_4 \oplus R_5
\end{aligned} \tag{19}$$

6. Empirical Results

First, we implemented the presented fault tolerant scheme in [6] and we tried to attack it by injecting faults according to the illustrated method in [5].We could find the secret key successfully in all of 1000 iterations. As described, the fault tolerant techniques are not always suitable to resist against fault attacks.

We implemented the illustrated scheme that uses the RS1 in the AES. Table 6 shows the results of insertion of ***Check_Redundancy*** and ***Correct_Redundancy*** of the RS1 to a prepared AES architecture. According to the results, area overhead for the ***AES_Check*** is more than the ***AES_Correct***. The ***AES_Correct*** can correct some of faults (see table 3). Thus, it is effective in fault tolerant implementations. Additionally, it can be used for fault attack tolerating. The illustrated attack was repeated on the both implementations; all injected faults were detectable by ***AES_Check***. Also, the generated faulty Ciphertexts by ***AES_Correct*** were not useful for fault attacks because they were not fitted on the fault model supposed.

Table 6. The required area for inserting *Check* and *Correct* modules in the AES architecture

	Slice	Flip Flop	LUT	BRAM
AES	485	561	889	20
AES_Check	2631	1140	4754	36
AES_Correct	838	804	1520	20

7. Conclusions

The difference between fault tolerance and fault attack tolerance methods has been described. We have shown that the powerful methods for making fault tolerant implementation (*e.g.*, [6]) do not counter the fault attacks. One technique which uses the inverse modules for fault detection with minimum time overhead and its implementation results have been presented. Three proposals for employing error correction codes have been presented in this paper. We introduced their implementation details and their fault coverage in correction and detection phases separately. The detailed comparison results of the inverse module methods, parity check techniques and our proposed methods have been illustrated. We have shown that the inverse module methods are the most cost-effective techniques if the inverse modules exist in the architecture; otherwise, the first proposed ECC (Reed-Solomon code with 8-bit redundancy for each byte) is an effective method in the fault tolerant implementations and in resisting against fault attacks. Additionally, the proposed techniques can be used in the key scheduling unit because the *KeyExpansion* module uses the XOR and the **S-box** operations and we have introduced the essential equations for supporting the redundancy parts of them.

References

[1] E. Biham, and A. Shamir, Differential Fault Analysis of Secret key Cryptosystems. In B. Kaliski, editor, *Advances in Cryptology - CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 513-525. Springer, 1997.

[2] J. Blömer, and J. P. Seifert, Fault Based Cryptanalysis of the Advanced Encryption Standard (AES). In *Financial Cryptography '03*, LNCS. Springer, 2003. Also available at http://eprint.iacr.org/,2002/075.

[3] G. Piret, and J. J. Quisquater, A Differential Fault Attack Technique against SPN Structures, with Application to the AES and Khazad. In *Cryptographic Hardware and Embedded Systmes – CHES 2003*, volume 2779 of *Lecture Notes in Computer Science*. Springer, 2003.

[4] C. Giraud, DFA on AES. Available at http://eprint.iacr.org/, 2003/008.

[5] P. Dusart, G. Letourneux, and O. Vivolo, Differential Fault Analysis on A.E.S. Available at http://eprint.iacr.org/, 2003/010.

[6] 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*, Volume 52, Number 4, pages 492-505, April 2003.

[7] R. Karri, W. Kaijie, P. Mishra, and K. Yongkook, Fault-Based Side-Channel Cryptanalysis Tolerant Rijndael Symmetric Block Cipher Architecture. In *Defect and Fault Tolerance in VLSI Systems (DFT '01)*, *Proceedings*, pages 418-426, 2001.

[8] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, and V. Piuri, Detecting and Locating Faults in VLSI Implementations of the Advanced Encryption Standard. In *Defect and Fault Tolerance in VLSI Systems (DFT '03)*, *Proceedings*, 2003.

[9] M. Karpovsky, K. J. Kulikowski, and A. Taubin, Differential Fault Analysis Attack Resistant Architectures for the Advanced Encryption Standard. Proc. *Smart Card Research and Advanced Applications (CARDIS)*, 2004.

[10] National Institute of Standards and Technology, Advanced Encryption Standard, NIST FIPS PUB 197, 2001.

[11] E. Berlekamp, R. Peile, and S. Pope, The Application of Error Control to Communications. *IEEE Communications Magazine*, Volume 25, Number 4, pages 44-57, April 1987.

[12] V. Bhargava, Forward Error Correction Schemes for Digital Communications. *IEEE Communications Magazine*, Volume 21, Number 1, pages 11-19, January 1983.

[13] A. Moradi, M.T. Manzuri, and M. Salmasizadeh, A Generalized Method of Differential Fault Attack against AES Cryptosystem. In *Cryptographic Hardware and Embedded Systems - CHES 2006*, volume 4249 of *Lecture Notes in Computer Science*, pages 91-100. Springer, 2006.

Amir Moradi received the B.S. degree in Computer Engineering from Shahid Beheshti University in Iran, in 2001. He also received the M.S. degree in Computer Engineering from Sharif University of Technology in Iran, in 2004. Currently, he is a Ph.D. candidate in Computer Engineering at the department of computer engineering of Sharif University of Technology. His research interests are in the areas of side channel attacks and the implementation of cryptographic algorithms.
E-mail: a_moradi@ce.sharif.edu

Mahmoud Salmasizadeh received the B.S. and M.S. degrees in Electrical Engineering from Sharif University of Technology in Iran, in 1972 and 1989 respectively. He also received the Ph.D. degree in Information Technology from Queensland University of Technology in Australia, in 1997. Currently he is an assistant professor in Electronic Research Center and adjunct assistant professor in Electrical Engineering Department at Sharif University of Technology, Tehran, Iran. His research interests include cryptography and network security. He is the founding member and the head of scientific committee, Iranian Society of Cryptology.
E-mail: salmasi@sharif.edu

Mohammad Taghi Manzuri-Shalmani received his B.S. and M.S. in Electrical Engineering from Sharif University of Technology (SUT), Iran, in 1984 and 1988, respectively. He also received the Ph.D. degree in Electrical and Computer Engineering from Vienna University of Technology, Austria, in 1995. Currently, he is an assistant professor in Computer Engineering Department of SUT, Tehran, Iran. His main research interests include digital signal processing, cryptography, image processing, and data communications.
E-mail: manzuri@sharif.com