

## Chapter 9: Fault Detection in Crypto-devices

Kaouthar Bouselam, Giorgio Di Natale, Marie-Lise Flottes, Bruno Rouzeyre

► **To cite this version:**

Kaouthar Bouselam, Giorgio Di Natale, Marie-Lise Flottes, Bruno Rouzeyre. Chapter 9: Fault Detection in Crypto-devices. Wei Zhang. Fault Detection, InTech, pp.177-194, 2010, 978-953-307-037-7. <10.5772/213>. <<http://www.intechopen.com/books/fault-detection/fault-detection-in-crypto-devices>>. <lirmm-00437252>

**HAL Id: lirmm-00437252**

**<https://hal-lirmm.ccsd.cnrs.fr/lirmm-00437252>**

Submitted on 26 Oct 2012

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Fault Detection in Crypto-Devices

K. BOUSSELAM, G. DI NATALE, M.-L. FLOTTES and B. ROUZEYRE  
*LIRMM (Université Montpellier II / CNRS UMR 5506)*  
*Montpellier, France*

This chapter presents a study on fault detection mechanisms involved in secure devices in order to prevent faults-based attacks. We explore the solutions based on the use of error detection codes (parity bits, CRC) and we discuss the strengths and the weaknesses of these solutions with regards to error and fault detection.

## 1. Introduction

Today's secure devices are mainly used for storage and processing of confidential data. Current products provide hardware and software secure solutions for civil and online identification, telecommunication, healthcare, banking, pay-TV, access control for restricted systems or areas, e-government... Tomorrow, they will include decision making capabilities for machine to machine applications. 20 billion of secure devices are forecasted in 2020 (4 billion in 2007) [Eurosmart, 2007].

Due to their applications, secure devices must be designed so that they can guarantee high levels of dependability and quality. But in addition to usual dependability features (reliability, availability, safety, robustness to environmental conditions), we also expect that they have the ability to protect information against unauthorized access and intentional misuse. The digital security mechanisms involved in such devices rely on various principles: secrecy of design and implementation, ciphering operations for encryption/decryption of confidential data, and hardware and software countermeasures for attack detection or tolerance.

Encryption is the process of transforming data in order to make it unreadable to anyone except those possessing the decryption key. Encryption can be symmetric or asymmetric. In symmetric encryption, a sender and a recipient share the same secret key, which is used for both encryption of plaintexts and decryptions of corresponding cipher texts. Conversely, the asymmetric algorithms use different keys for encryption and decryption. A sender  $S$  communicate his/her public key to all recipients  $R$ ; messages encrypted with that public key by any  $R$  can only be decrypted by the sender  $S$  using his/her corresponding private key. The symmetric and asymmetric algorithms are generally public, while the secret and private keys are kept secret. Symmetric encryption is fast but senders and recipients need to define a "secure" key exchange process prior to start communication. Asymmetric encryption can be used for exchanging this secret key at the beginning of a communication, and then a symmetric-key algorithm using that secret key can be used for fast encryption during the remainder of the communication. The *section 2* details the Advanced Encryption Standard

(AES), a symmetric encryption process based on the Rijndael algorithm from Joan Daemen and Vincent Rijmen.

Due to their applications, secure devices are subject to attacks aiming to gather private information. Discovering the secret key of a symmetric cipher for instance allows decrypting the text encrypted with that key. Numerous types of attack rely on the hardware implementation of the cipher since cryptanalysis on recent algorithms is hopefully not practical. Most recent invasive attacks using probes or modifications of the chip are powerful but destroy the package, require time of specialists in laboratories and a proper budget. Non-invasive side-channel attacks use leakage information related to the processed data such as the operational timing, the power consumption of the chip, or the electromagnetic interferences of signals. Active but semi or non-invasive fault-based attacks rely on perturbation of the circuit behavior and use (expected) production of erroneous results for inferring secret information. *Section 3* gives an overview of implementation techniques and data analysis performed on the reference encryption standard AES for fault-based attacks.

Cipher algorithms are often integrated as coprocessors for better performance. As any other function implemented in hardware, these coprocessors need to be carefully tested in order to determine whether they are capable of performing the intended functions. So, test contributes to the dependability and the quality of the devices in the sense that it prevents insertion of failing hardware in dependable devices, and allows revealing faulty behaviours during the chip lifetime. Classically, targeted fault-set includes permanent (or intermittent) faults, which model physical defects due to manufacturing defaults or aging, and “natural” transient faults due to the environment (particle hit) (Reed et al., 2003). However, test can also contribute to digital security by preventing maliciously injected transient faults to contribute in revealing confidential data. While the detection of permanent faults is generally performed during the execution of specific test modes after production or during maintenance times, transient fault detection must be concurrent to the mission mode of the circuit. Online test solutions are thus implemented in secure chips as part of protection mechanisms targeting fault-attacks. In this context, fault-set includes transient stuck-at faults on gate signals in the combination part of the cipher, and bit-flips on memory elements (Leveugle, 2007). In this context, potential faults are generally detected through identification of erroneous signal(s) on a function or sub-function output. *Section 4* presents the solutions from the literature for error-detection schemes in AES ciphers. These solutions are first compared using common evaluation criteria such as implementation cost and fault detection latency.

These solutions are then compared in terms of error detection capacity in *Section 5*. Experimental data shows the percentage of undetected errors for each scheme according to the number of erroneous bits injected in one or several bytes during execution of the AES encryption.

Finally, the quality of the protection mechanisms is also evaluated in terms of capacity to detect most frequent misbehavior. *Section 6* presents an analysis on the diffusion of errors in the responses of the AES cipher in the case of single transient stuck-at faults and discuss the choice of appropriate detection mechanisms.

## 2. Advanced Encryption Standard

Even if fault detection techniques described in this chapter are general enough to be implemented for secure devices implementing any cryptographic algorithm, we use the Advanced Encryption Standard (AES) as support example. This section details the AES which was adopted by the US government (FIPS-197, 2001).

The AES algorithm is a symmetric block cipher that can encrypt (encipher) and decrypt (decipher) information. Encryption converts data to an unintelligible form called cipher text; decrypting the cipher text converts the data back into its original form, called plaintext. Encryption and decryption are performed by means of the same cryptographic secret key. The AES algorithm is capable of using cryptographic keys of 128, 192, and 256 bits to encrypt and decrypt data in blocks of 128 bits. This section focuses on the encryption algorithm for 128-bits cryptographic keys (details on decryption and others key lengths are fully described in (FIPS-197, 2001)).

The basic unit for processing in the AES algorithm is a byte. Input, output and secret key bit sequences are internally processed on a two-dimensional array of bytes called the *State*. The State consists of four rows of four bytes. In the State array, denoted by the symbol  $s$ , each individual byte has two indices, with its row number  $r$  in the range  $0 \leq r < 3$  and its column number  $c$  in the range  $0 \leq c \leq 3$ .

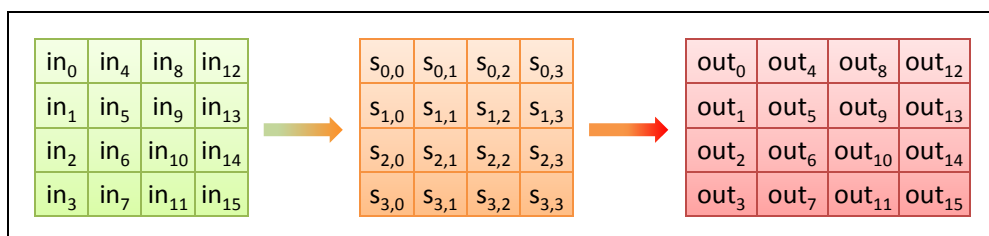


Fig. 1. Input, State and Output arrays

The AES algorithm is based on permutations and substitutions. Permutations are rearrangements of data, while substitutions replace one unit of data with another. AES performs permutations and substitutions using several different functions.

At the beginning of the encryption, the input is copied to the State array using the conventions described in Fig. 1. After an initial XOR operation between the State array and the cipher key, the State array is transformed by implementing a round function that is repeated 10 times (see Section 2.1), with the final round differing slightly from the first 9 rounds. The final State is then copied to the output as shown in Fig. 1.

The round function is parameterized using a Key Expansion function that generates a variation of the original cipher key at each round repetition (see Section 2.2).

Fig. 2 summarizes the AES algorithm.

### 2.1 Round

Round function is composed of 4 operations: SubBytes, ShiftRows, MixColumns, and AddRoundKey. These functions operate and modify the value of the State. As shown in Fig.

2, all rounds are identical with the exception of the final round, which does not include the MixColumns transformation.

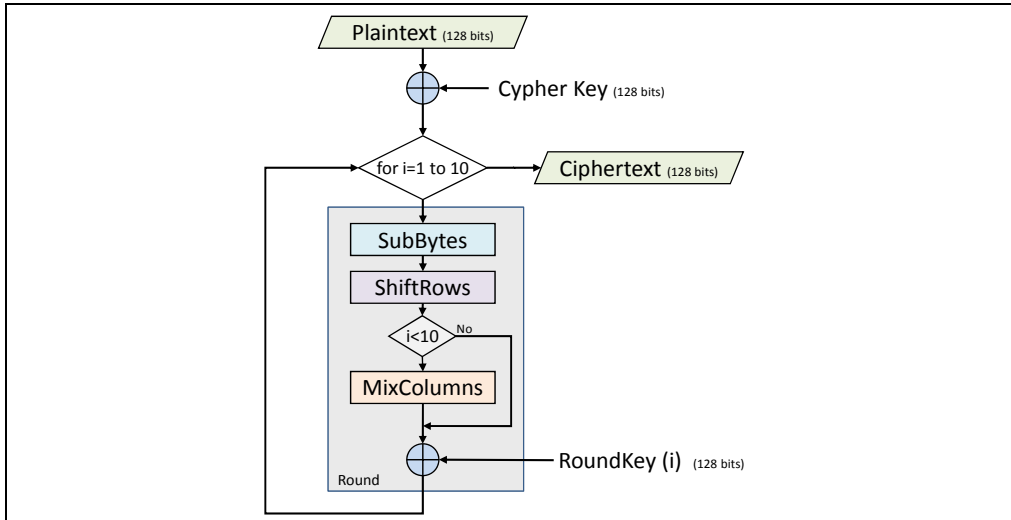


Fig. 2. AES Algorithm

A schematic view of the Round function is shown in Fig. 3. Next paragraphs describe in detail each of the 4 operations.

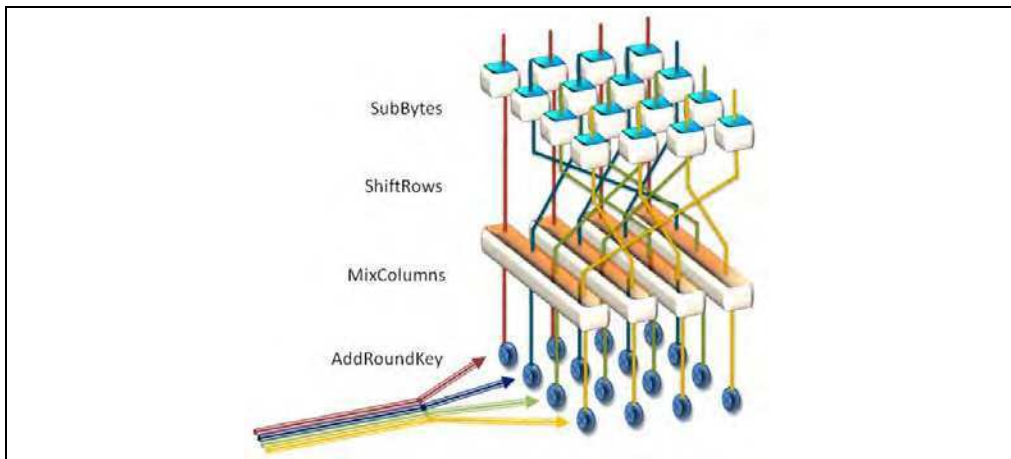


Fig. 3. AES Round

The **SubBytes** transformation is a non-linear byte substitution that operates independently on each byte of the State using a substitution table (S-box). The S-box is constructed by composing two transformations:

1. The multiplicative inverse in the finite field  $GF(2^8)$ , where the element  $(00000000)_2$  is mapped to itself;
2. The following affine transformation (over  $GF(2)$ ):

$$b'_i = b_i \oplus b_{(i+4)\text{mod}8} \oplus b_{(i+5)\text{mod}8} \oplus b_{(i+6)\text{mod}8} \oplus b_{(i+7)\text{mod}8} \oplus c_i$$

for  $0 \leq i \leq 7$ , where  $b_i$  is the  $i^{\text{th}}$  bit of the byte, and  $c_i$  is the  $i^{\text{th}}$  bit of a byte  $c$  whose value is fixed by the standard and it is equal to  $\{01100011\}$ .

The S-box can be seen as a look-up table where for each 8-bits input there is a unique 8-bits output value. Several hardware implementations have been proposed in the literature. They can be classified in three major categories:

1. ROM-based (Zhang and Parhi, 2002);
2. Look-up table implementation (combinational logic): the S-box is written in any hardware description language as a look-up table and it is converted into a gate-level circuit using a logic synthesizer (Di Natale et al., 2007 A). This solution allows the best performances in terms of speed and dynamic power consumption;
3. Mathematical implementation: instead of describing the S-box as look-up table, the circuit is described as the sequence of operations in the finite field  $GF$  and then synthesized. This alternative can provide better solutions in terms of area occupation and leakage power consumption (Wolkerstorfer et al., 2002).

In order to provide an idea of the differences between look-up table based implementations and mathematical implementations, we implemented in VHDL the solutions proposed in (Di Natale et al., 2007 A) (look-up table) and in (Wolkerstorfer et al., 2002) (mathematical) and then we synthesized the two circuits. Table 1 shows the comparison of area, speed and power consumptions obtained using a 90nm technology library provided by ST Microelectronics, where a new input is provided to the circuit every 3 ns.

	<i>Look-up table implementation</i>	<i>Mathematical implementation</i>
Area [ $\mu\text{m}^2$ ]	1993	1122
Delay of critical path [ns]	1.25 (14 logic levels)	2.45 (25 logic levels)
Average Dynamic Power [mW]	0.136	0.907

Table. 1. Comparison of area, delay and power between look-up table implementation and Mathematical implementation (ASIC)

The **ShiftRows** operation changes the byte position in the State. It rotates each row with different offsets to obtain a new State as follows:

$$State = \begin{bmatrix} S_0 & S_4 & S_8 & S_{12} \\ S_5 & S_9 & S_{13} & S_1 \\ S_{10} & S_{14} & S_2 & S_6 \\ S_{15} & S_3 & S_7 & S_{11} \end{bmatrix}$$

The first row is unchanged; the second row is rotated one byte position to the left, the third row two byte positions, and the fourth row three byte positions. ShiftRows is a linear transformation. Hardware implementation of this transformation resumes to wiring.

The **MixColumns** operates column-wise altering all the bytes of the same column. It treats a column as a 3rd degree polynomial with coefficients in  $GF(2^8)$  and produces the new

column by multiplying it with a constant polynomial. This operation is performed modulo a 4th degree polynomial with coefficients in GF(2<sup>8</sup>).

Let s<sub>i</sub>,s<sub>i+1</sub>,s<sub>i+2</sub>,and s<sub>i+3</sub> be four consecutive bytes (column-wise), i∈{0,4,8,12} in the State matrix before MixColumns. These four bytes are transformed as:

$$\begin{bmatrix} t_i \\ t_{i+1} \\ t_{i+2} \\ t_{i+3} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_i \\ s_{i+1} \\ s_{i+2} \\ s_{i+3} \end{bmatrix}$$

where the t<sub>i</sub>'s are the State bytes after the MixColumns operation. The MixColumns operation is typically implemented as XOR-trees.

The **AddRoundKey** adds the corresponding round key to the current State. In the GF(2<sup>8</sup>) field, addition is implemented as a bit-wise XOR operation between the two elements. The implementation of the AddRoundKey operation resumes to a single layer of XOR gates.

**2.2 Key Expansion**

The AES algorithm takes the Cipher Key and performs a Key Expansion routine to generate a key schedule. The Key Expansion generates a total of 11 words (the initial key plus one roundkey for each of the ten rounds). Each generated key has 128 bits that are organized in a state array as defined for the data in Figure 1 (i.e., the j<sup>th</sup> byte of the key state word is composed of the bits from 8•j to 8•j+7).

We define K<sub>i</sub> the key of i<sup>th</sup> round (0 ≤ i ≤ 10), where K<sub>0</sub> denotes the initial secret key. The generation of a new round key depends on the previous one. Fig. 4 shows the scheme of generation on a new key K<sub>i+1</sub> starting from K<sub>i</sub>. The transformation F<sub>i</sub> is a non-linear function composed of 4 S-boxes (one for each byte), a rotation of bytes, and the addition of a round specific constant, defined by the standard. More details about the generation process are given in (FIPS-197, 2001).

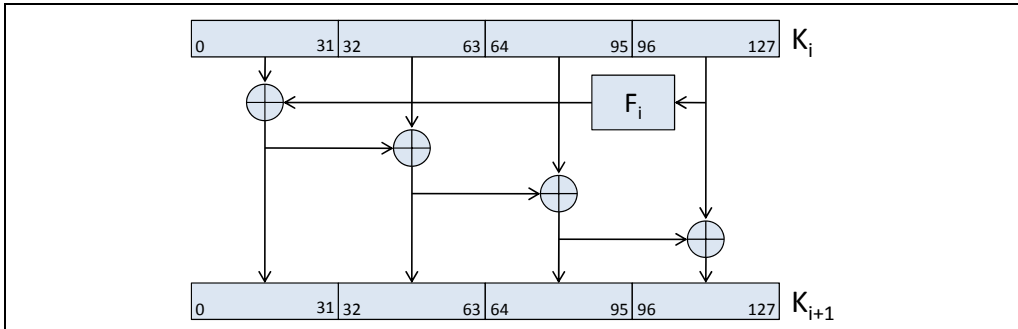


Fig. 4. AES Key Expansion Scheme

**3. Fault attacks**

As mentioned in the introduction, faults can be either permanent (due to fabrication defects) or transient (due to environmental conditions such as the exposure to cosmic radiations). For the particular case of crypto-coprocessors, transient faults can also be maliciously and intentionally injected in the circuit in order to retrieve the secret key (Blömer and Seifert,

2003). By comparing the result of a normal encryption with a faulty one, a hacker can deduce the key. These techniques are referred to as "Differential Fault Analysis" (DFA). Faults can be injected by different means such as temperature variation, clock frequency modification, exposure to radiations UV, X or visible light (Kim and Quisquater, 2007). The formers are quite efficient for software implementations of the crypto algorithm, while the latter (i.e., laser-based fault injection) is particularly well-suited for hardware implementations. The main advantage of laser-based fault injection is the localization of the fault (in the timing and the spatial domains).

In this context, it is thus of prime importance to be able to detect such faults. Nevertheless, independently of the mean used to inject the fault, the induced error must satisfy certain conditions in order to be successfully exploited. The following subsections quickly review the published DFA attacks on AES and analyze these conditions on the injected errors.

### 3.1 Attacks on 1 bit

One of the first cryptanalysis method using faults on AES has been published by (Giraud, 2005). The considered error is a single faulty bit on any of the 128 State bits at the input of the SubBytes operation during execution of the last encryption round (10<sup>th</sup> round). This error of multiplicity one (only one bit is affected) is equivalent to an error of multiplicity one appearing on the round key or on the input of the previous AddRoundKey operation. This error spreads, affects the output of the SubBytes operation and, since the last round does not include the Mixcolumn operation, it also affects one (and only one) byte of the final Output array. The attack can be summarized as follows: let denote  $B_9$  one of the output bytes of the last but one round (i.e., the one where the fault will be injected),  $e$  the 8-bit error mask (it contains only one asserted bit in the position where the error is considered), and  $BK_{10}$  the byte of the round key  $K_{10}$  at the same position of  $B_9$  (among the 16 bytes). The correct output result should be  $\{SubBytes(B_9) \oplus BK_{10}\}$ . Because of the fault, the result is  $\{SubBytes(B_9 \oplus e) \oplus BK_{10}\}$ . By adding these two results, it comes:

$$d = \{SubBytes(B_9) \oplus BK_{10}\} \oplus \{SubBytes(B_9 \oplus e) \oplus BK_{10}\} = SubBytes(B_9) \oplus SubBytes(B_9 \oplus e)$$

There are 8 possibilities for the faulty bit. For each possibility, the list of possible  $B_9$  values is built up. The same process is iterated with other values of  $e$ . In average, the value of  $B_9$  can be obtained with 3 faults. Then, by adding  $SubBytes(B_9)$  to the correct result, the byte  $BK_{10}$  of the key is obtained. In about 50 fault injections, the whole round key  $K_{10}$  can be fully determined. The primary key  $K_0$  can be mathematically derived from  $K_{10}$ .

In (Blömer and Seifert, 2003) the authors proposed attacks based on the "safe error" principle, i.e. "the error affects the result or not". The considered fault is a stuck-at-0 affecting one bit of the key. If the result is faulty, the actual value of the key bit is 1.

In (Blömer and Krummel, 2006) the authors report another attack based on the injection of an error of multiplicity one which exploits collision effects, i.e. the fact that a two messages (one without, the other one with an error) will give the same result.



### 3.2 Attacks on 1 byte

In (Giraud, 2005), Giraud also proposes a more complex attack with the advantage of considering a less restrictive error model than errors on a single bit only. Here, errors of multiplicity  $x$  ( $x \geq 1$ ) affecting one byte are taken into account. The attack is composed of three steps:

1. an error is injected on the round key  $K_9$  just before the last KeySchedule. This error affects 5 bytes of the next key  $K_{10}$ , four of them being on the same line of the Key State array. Using the same set of equations than in the previously mentioned attack, one byte of the last column of the last round key is obtained. By repeating this step 4 times, the 4 bytes of the last column of  $K_9$  are retrieved.
2. an error is injected on the last column of  $K_8$  before the penultimate KeySchedule. This spreads on 10 bytes. One of these faulty bytes gives information on the propagated error during the last KeySchedule. Since, the last column of  $K_9$  is known, 3 bytes of  $K_8$  can be definitely determined and it remains 130 possibilities of its fourth byte. This gives the penultimate column of  $K_9$ .
3. in the last step, an error is injected on the last column of the State array before round 10. Thus, all values which satisfy the observed differences on the output are determined using an exhaustive search on the 242 possibilities. Since, the last column before round 10 is known and also the two last columns of  $K_9$ , one can derive 14 bytes of  $K_{10}$ . The two last ones are determined using an exhaustive search.

Some other attacks relying on several fault injections on bytes are reported in (Chen and Yen, 2003), (Piret and Quisquater, 2003), and (Dusart et al., 2003).

The attack in (Blömer and Seifert, 2003), based on the "safe error" principle, can be extended to the case of attack on 1 byte. In this case all the input bits of an S-box must be stuck at 0. Then, by applying all 256 values at the input of the S-box, a collision appears: when the byte has the same value as the key, the result equals the one obtained with the stuck-at fault.

### 3.3 Attacks on several bytes on the same column

In (Moraidi et al., 2006) the authors propose a generalization of the attack on a single byte, focusing on 3 or 4 bytes in the first State array column. The faults can be injected at any step of the AES process. For the attack to be successful, it is necessary to know if the error affects 3 or 4 bytes. Furthermore, 6 and 1500 error injections have to be performed for 3 and 4 faulty bytes respectively.

### 3.4 Conclusions

In this paper, we do not discuss about the actual capabilities of injecting faults according to the hypothesis underlying those attacks. Nevertheless, from this overview, it can be concluded that:

- To our knowledge, errors affecting more than one byte (except (Moraidi et al., 2006)) cannot be exploited to perform an attack. Thus it's of prime importance to detect errors located within a byte. All error multiplicities (1 to 8) have to be considered.

- According to the variety of reported attacks, all time steps (rounds) of the AES algorithm are prone to fault injections. Thus, the data protection mechanism must span over the whole AES process.
- While errors affecting more than one byte are not exploitable by nowadays reported attacks, their detection is of interest since, firstly it helps in detecting an attack (for instance laser-based attacks need in practice many shots before succeeding in flipping bits within a single byte), and secondly the ingenuity of hackers may make efficient these attacks in the future.

#### 4. Error Detection Methods

Concurrent fault detection for hardware implementations of secure devices is important not only to protect the system from random faults, but also to protect it from an attacker who may maliciously inject faults in order to find the encryption secret key.

Several error detection solutions have been proposed in the literature, they rely on the use of some form of spatial or temporal redundancy. (Karri et al., 2002) propose a solution suitable for systems containing both encryption and decryption modules. The basic idea is to use the decryption module to decrypt the encrypted data (ciphertext), and to verify that the ciphertext is equal to the actual plaintext. (Maistri et al., 2007) propose a design solution that exploits temporal redundancy by a Double-Data-Rate mechanism in order to perform the computation twice without affecting adversely the overall throughput of the system. (Monnet et al., 2006) propose quasi-delay insensitive (QDI) asynchronous logic design to cope with fault attacks. Indeed, specific properties of asynchronous circuits make them inherently resistant against a large class of faults. Finally, several papers have been published on the use of codes to detect faults in secure devices. In particular, we focus the analysis on the architectures proposed in (Wu et al., 2004), (Bertoni et al., 2003), (Yen and Wu, 2006), and (Di Natale et al., 2007 B), as representative of the wide range of code-based solutions.

In a code-based fault detection scheme for a block of logic gates, the basic approach is that the output code of the block is predicted from the input data and this code is compared to the actual code calculated from the block output data of the block. The disadvantage of this technique is the use of the data bits to predict the code since corrupted bits may affect the code and the error may not be detected.

(Wu et al., 2004) propose the use of a parity bit for each of the 16 S-boxes that are XOR-merged in a single parity bit for the whole 128-bits data block. The single parity bit is used to check the correctness of the other blocks of the round. The datapath implementing the AES round with concurrent checking is shown in Fig. 5.

The principle is to compare the input parity of a round with the predicted output parity of the previous round, at each round. For that, the parity of the input values of a round is first determined ( $P(x)$  in the figure). Then, the parity bit is modified according to the operations executed during the round. In particular, a parity bit of the word composed by the input ( $x_i$ ) and the output ( $y_i$ ) is predicted for each S-box. The 16 parity bits so calculated are then added to obtain a single parity bit ( $P(x) \oplus P(y)$ ). This parity bit is added again to the input parity bit  $P(x)$  so that the parity  $P(y)$  is determined.  $P(y)$  is not affected by neither ShiftRows, which only changes the byte position in the state array, nor MixColumns operations (Wu et al., 2004). This parity bit is then modified by adding the parity of the key  $P(k)$  and stored in a

1-bit register. Errors are detected when this predicted parity bit differs from the input parity computed at the beginning of the next round.

The authors implemented this solution on a Xilinx Virtex 1000 FPGA. The hardware overhead is about 8% and the additional time delay is about 5%. Details about error detection capability are given in Section 5. It must be noted that this technique, while being effective when S-boxes are ROM-implemented, is quite costly for standard gates implementations.

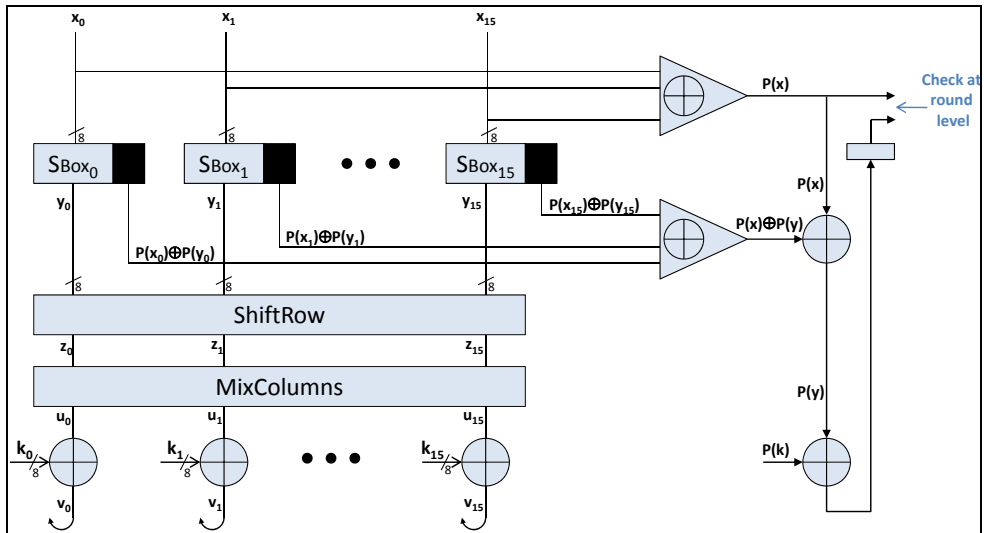


Fig. 5. (Wu et al., 2004) concurrent checking scheme

In (Bertoni et al., 2003) the authors propose to use 16 parity bits instead of a single bit. In particular, a parity bit is associated to each byte of the state (see Fig. 6).

Concerning the S-boxes, the authors propose a ROM-based implementation. The extra parity bit is stored in that ROM resulting in a  $256 \times 9$  bits memory. Moreover, to detect input parity errors and some internal memory (data or decode) errors, authors propose the use of a  $512 \times 9$  ROM, where the ninth bit is driven by the input parity bit. They deliberately force all the ROM words corresponding to a wrong input address (i.e., S-boxes input with a wrong parity bit associated) with a dummy output value where the output parity is wrong, so that the detection mechanism will detect the fault.

As before, the parity bit associated to each byte does not change after the ShiftRows operation. On the contrary, the prediction of the output parity bits of the MixColumns is the most mathematically complex for this type of architecture, because it depends on the value of 4 bytes of the state (see details in (Bertoni et al., 2003)). For AddRoundKey operation, the prediction of the output parity bit easily consists in adding the current parity bits with the parity bits of the corresponding round key.

(Yen and Wu, 2006) propose the use of a systematic  $(n+1, n)$  Cyclic Redundancy Check (CRC) over  $GF(2^8)$  to detect errors during encryption, where  $n \in \{4, 8, 16\}$  is the number of bytes contained in the message (see Fig. 7). The generator polynomial is  $g(x) = 1 + x$ . The CRC

byte can be associated either to each column of the state (i.e., to 4 bytes and so they use CRC(5,4)), or to two columns (i.e., 8 bytes, so CRC(9,8)), or to the whole State (CRC(17,16)). This solution allows a very high error detection level at the cost of high area overhead (in the order of a thousand additional ports).

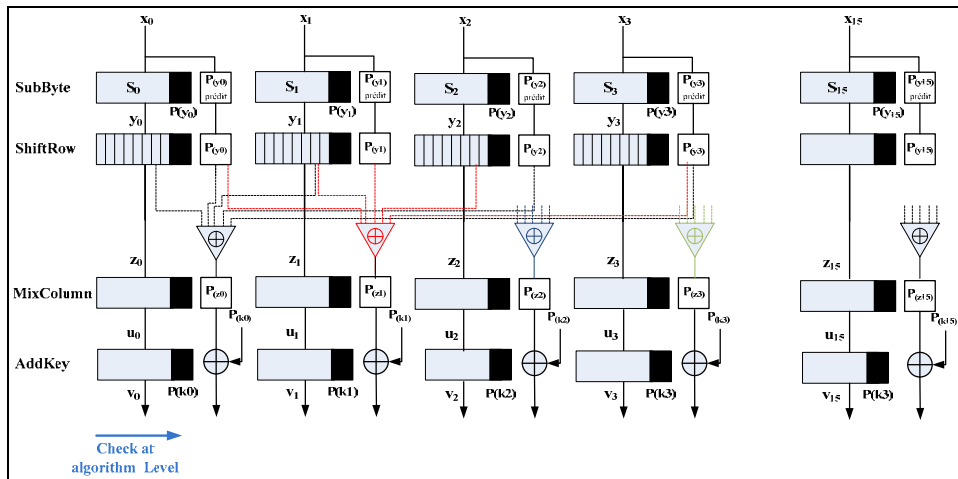


Fig. 6. AES round with (Bertoni et al., 2003) model

Since S-boxes represent the largest part of the circuit, (Di Natale et al., 2007 B) propose a solution that focuses on S-boxes only. They propose the use of two parity bits per S-box, one parity bit for the input byte of the S-box and one for its output byte. This double parity per S-box was also proposed in (Wu et al., 2004), but the two parity bits are now independently generated by dedicated prediction logics.

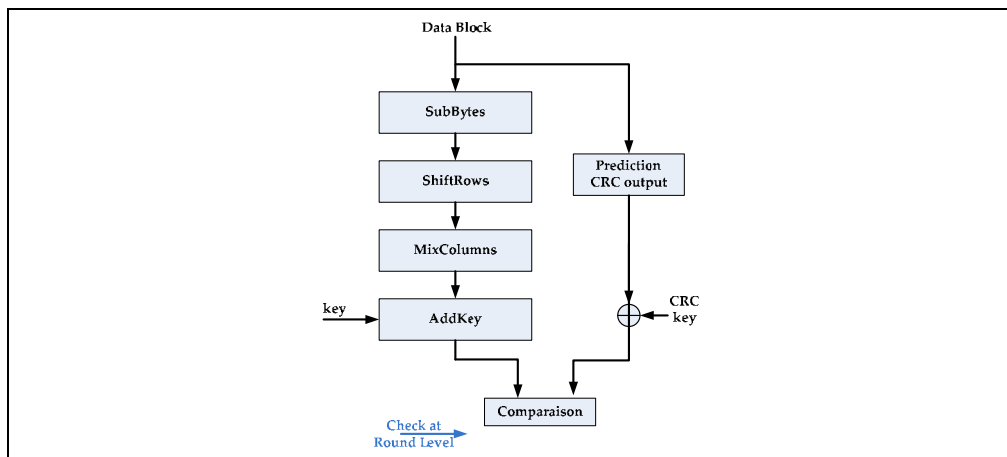


Fig. 7. AES round with (Yen and Wu, 2006) model

Then, the actual output parity is compared with the predicted output parity bit, and the actual input parity bit is compared to the predicted one (Fig. 8). When the S-box and the prediction circuits are synthesized as combinational logic, the area overhead is 38.33% with respect to the original S-box. This double parity checking allows additional detection of 27% of errors of even multiplicity compared to the solution of (Wu et al., 2004), besides all odd multiplicity of errors.

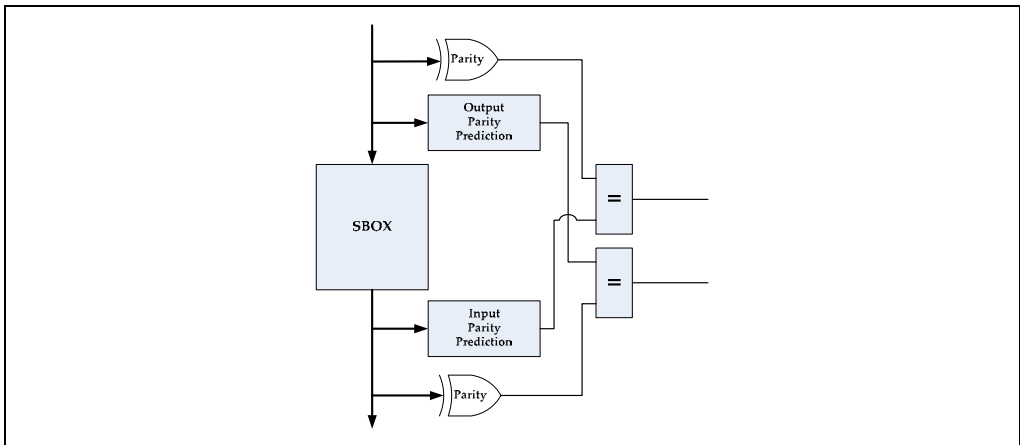


Fig. 8. SubBytes transformation with (Di Natale et al., 2007 B) model

Concerning error detection flags, the predicted and actual parity bits can be compared after each operation, each round, or after execution of the whole encryption. These solutions are equivalent from the detection capability point of view. They only differ in terms of hardware overhead and detection latency. Multiplicity of checkpoints decreases the error detection latency but increases the hardware overhead.

## 5. Error Detection

The error detection schemes detailed in the preceding section are now compared in terms of error detection capabilities.

Concerning the experimental setup, we assume single transient stuck-at faults, which are supposed to occur during the execution of one operation in one round (Fig. 9). S-boxes are implemented using random logic. Concerning the scheme presented in (Yen and Wu, 2006), we used a CRC (5,4) i.e. 4 CRC bytes, one for each column of the State array (32 bits). Since the error detection scheme presented in (Di Natale et al, 2007 B) only addresses errors on the S-boxes outputs, this protection scheme is completed with the solution proposed in (Bertoni et al., 2003): a faulty parity bit is affected to an S-box operation when either faulty input or output parities are detected. This S-box parity bit is subsequently used in the following prediction operations.

We first analyzed the detection capabilities of these four techniques with respect to errors affecting a single byte. Errors have been exhaustively injected in every byte, every operation and at every round. Table 2 reports the error multiplicity (one to eight faulty bits), the

number of simulated errors for each error multiplicity (e.g.  $16 * 40 * \binom{2}{8} = 17920$  error instances for 2 faulty bits among 8 bits, affecting one byte over 16 in the State array, after one of the 40 operations executed during the encryption), and the number of the undetected errors for each technique under evaluation.

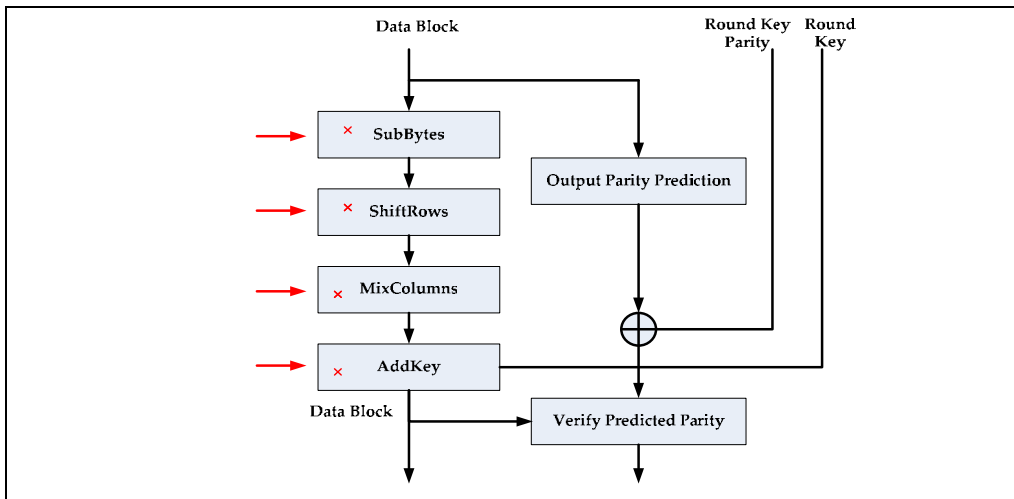


Fig. 9. Errors injection model

As expected, errors with odd multiplicity are more easily detected with detection schemes based on parity codes (Wu et al., 2004), while the CRC-based scheme proposed in (Yen and Wu, 2006) outperforms other techniques by detecting even-multiplicity errors too.

Output errors multiplicity	# of possible errors	Number and percentage of the undetected errors			
		(Wu et al., 2004) (1 parity bit)	(Bertoni et al., 2003) (16 parity bits)	(Yen and Wu, 2006) (4 CRC bytes)	(Di Natale et al., 2007 B) (16 parity bits + S-box protection)
1	5120	125 (2.44%)	0 (0%)	0 (0%)	0 (0%)
2	17920	2644 (14.75%)	16912 (94.38%)	0 (0%)	14666 (81.84%)
3	35840	841 (2.35%)	0 (0%)	0 (0%)	0 (0%)
4	44800	6614 (14.76%)	39760 (88.75%)	0 (0%)	34095 (76.10%)
5	35840	898 (2.51%)	0 (0%)	0 (0%)	0 (0%)
6	17920	2620 (14.62%)	14896 (83.13%)	0 (0%)	12642 (70.55%)
7	5120	110 (2.15%)	0 (0%)	0 (0%)	0 (0%)
8	640	98 (15.31%)	496 (77.50%)	0 (0%)	171 (26.72%)

Table 2. Detection capability for errors in a single byte

In the next experiments we injected random errors affecting any of the 128 state bits, with error multiplicity ranging from 1 to 64 faulty bits. Note that if errors affecting several bytes are not easily exploitable during DFA, their detection is of prime interest for detecting the

attack itself. For each error multiplicity, 1000 randomly-chosen injection positions have been selected among randomly-chosen State array bits after execution of a randomly-chosen encryption operation. Simulation results are reported in Table 3.

All the techniques detect errors with multiplicity larger than 6 bits, except the detection scheme in (Wu et al., 2004) that provides only one parity bit for the whole State array.

Error multiplicity	Number and percentage of the undetected errors			
	(Di Natale et al., 2007 B) (16 parity bits + S-box protection)	(Yen and Wu, 2006) (4 CRC bytes)	(Bertoni et al., 2003) (16 parity bits)	(Wu et al., 2004) (1 parity bit)
1	0 (0.00%)	0 (0.00%)	0 (0.00%)	975 (2.44%)
2	1883 (4.71%)	549 (1.37%)	2164 (5.41%)	5992 (14.98%)
3	0 (0.00%)	2 (0.01%)	0 (0.00%)	1037 (2.59%)
4	262 (0.66%)	70 (0.18%)	320 (0.80%)	6069 (15.17%)
5	0 (0.00%)	1 (0.00%)	0 (0.00%)	981 (2.45%)
6	61 (0.15%)	0 (0.00%)	80 (0.20%)	6050 (15.13%)
7	0 (0.00%)	0 (0.00%)	0 (0.00%)	995 (2.49%)
8	0 (0.00%)	0 (0.00%)	0 (0.00%)	5956 (14.89%)
9	0 (0.00%)	0 (0.00%)	0 (0.00%)	1022 (2.56%)
10	0 (0.00%)	0 (0.00%)	0 (0.00%)	5933 (14.83%)
11	0 (0.00%)	0 (0.00%)	0 (0.00%)	991 (2.48%)
12	0 (0.00%)	0 (0.00%)	0 (0.00%)	6021 (15.05%)
13	0 (0.00%)	0 (0.00%)	0 (0.00%)	1001 (2.50%)
14	0 (0.00%)	0 (0.00%)	0 (0.00%)	6021 (15.05%)
15	0 (0.00%)	0 (0.00%)	0 (0.00%)	1018 (2.55%)
16	0 (0.00%)	0 (0.00%)	0 (0.00%)	5997 (14.99%)
....				
63	0 (0.00%)	0 (0.00%)	0 (0.00%)	1019 (2.55%)
64	0 (0.00%)	0 (0.00%)	0 (0.00%)	6008 (15.02%)

Table 3. Detection capability for random errors in 1 to 64 bits of a State

## 6. Faults and Error multiplicity

In the previously mentioned papers, the authors present the error detection capability of the methods (for instance, parity based architectures can detect all odd multiplicity errors). However, none of the papers correlate the error multiplicity, that appear at the output of round operations, with detected transient stuck-at faults affecting signals in logic blocks during execution of these operations.

In this section we evaluate the effectiveness of the solutions described in Section 4 with respect to fault detection. We focused this study on the S-box only, for the following reasons:

- it implements a 8 input bits function so it is possible to perform an exhaustive study;
- most of the attacks are performed on this block;
- Shiftrow module is anyway out of concern since it resumes to wiring;

- AddRoundKey resumes to a single layer of XOR gates. Thus, under the selected fault model, only error of multiplicity 1 may occur when a fault affects a XOR gate signal;
- MixColumns operates on 32 input bits so an exhaustive study cannot be performed.

Moreover, since the error propagation on the output of the SubBytes operation strongly depends on the implementation (netlist) of the S-box, we implemented different versions of the S-box using different synthesis parameters and implementation styles. In particular, we implemented the following designs using the AMS 0.35 $\mu$ m technology library:

- Sbox1: description in VHDL as combinational look-up table, synthesis with Cadence<sup>®</sup>, 553 cells
- Sbox2: description in VHDL as combinational look-up table, synthesis with Design Compiler (Synopsys<sup>®</sup>) with “-map\_effort high” option, 477 cells
- Sbox3: description in VHDL as combinational look-up table, synthesis with Design Compiler with “-map\_effort medium” option, 482 cells
- Sbox4: description in VHDL as combinational look-up table, synthesis with Design Compiler with “-map\_effort low” option, 474 cells
- Sbox5: mathematical description in VHDL of two blocks: the inversion in GF(2<sup>8</sup>) described as combinational look-up table plus the affine transformation, synthesis with Design Compiler with “-map\_effort high” option, 481 cells
- Sbox6: mathematical description in VHDL, by using the decomposition of calculations in GF(2<sup>4</sup>) as described in (Wolkerstorfer et al., 2002), synthesis with Design Compiler with “-map\_effort high” option, 193 cells

For each S-box implementation we performed exhaustive fault simulation, i.e. we applied all the possible input values (256 values) and we fault simulated the behavior of the device for each possible stuck-at in the circuit. The fault simulation produced a fault dictionary composed by all the possible couples C formed by { input value / fault } and, associated to each of them, the number of erroneous bits at the output of the S-box.

Table 4 reports the number of couples C leading to error multiplicities ranging from 0 to 8 for each S-box implementation. The first column (error multiplicity = 0) represents all the cases where the fault is not excited by a specific input value, or its effect is not propagated to the output. For all the other cases, the cell reports, besides the number of couples C, the percentage with respect to the overall number of couples and, in bold, the percentage with respect to the number of couples that lead to at least 1 error at the output of the S-box.

It is possible to note that, for instance, among the 11% of fault simulations for which a fault injection results in erroneous data on the Sbox1 output (89% of experiments result in 0 error), about 78% of the cases result in only one erroneous output bit, justifying therefore the use of code-based solution that exploits simple parity bit. Likewise, the 14 couple [fault/vector] providing 8 output errors correspond to faults at the input of the Sbox. It might not be necessary to protect the Sbox with detection schemes able to detect 8 error bits, if these errors are detected at the input of the Sbox.

Obviously, different S-box implementations lead the series of fault injection to different profiles in terms of error multiplicity. The presented implementations differ in terms of power consumption, performance and area according to chosen synthesis parameters. But the synthesis tool generates also quite different implementations whether the initial VHDL model is described as a look-up table or a mathematical expression. When starting from look-up table, the number of errors at the output of the S-box is concentrated around 1 or 2 erroneous bits. On the contrary, mathematical-based architectures (Sbox5 and Sbox6)



are composed of several blocks that operate in cascade, and an error in a particular element is spread over several output bits. Therefore this type of implementation is first more sensitive to faults (in the case of Sbox6, 37% of couples generates an error at the output, while only 11% for Sbox1), and second it generates a higher number of output errors (the highest average is between 4 and 5).

Such experiments must be conducted for selecting appropriate fault detection schemes.

	0	1	2	3	4	5	6	7	8
Sbox1	450382 90%	36551 7% / <b>78%</b>	6448 1% / <b>14%</b>	2033 0% / <b>4%</b>	953 0% / <b>2%</b>	474 0% / <b>1%</b>	235 0% / <b>1%</b>	62 0% / <b>0%</b>	14 0% / <b>0%</b>
Sbox2	441836 89%	23664 5% / <b>43%</b>	16744 3% / <b>30%</b>	8547 2% / <b>15%</b>	3922 1% / <b>7%</b>	1631 0% / <b>3%</b>	637 0% / <b>1%</b>	147 0% / <b>0%</b>	24 0% / <b>0%</b>
Sbox3	438510 89%	23849 5% / <b>44%</b>	16609 4% / <b>30%</b>	8212 2% / <b>15%</b>	3801 1% / <b>7%</b>	1544 0% / <b>3%</b>	619 0% / <b>1%</b>	145 0% / <b>0%</b>	23 0% / <b>0%</b>
Sbox4	441855 89%	23658 5% / <b>43%</b>	16745 3% / <b>30%</b>	8534 2% / <b>15%</b>	3924 1% / <b>7%</b>	1629 0% / <b>3%</b>	636 0% / <b>1%</b>	147 0% / <b>0%</b>	24 0% / <b>0%</b>
Sbox5	418618 86%	7311 2% / <b>11%</b>	8973 2% / <b>13%</b>	4808 1% / <b>7%</b>	9949 2% / <b>15%</b>	27511 6% / <b>41%</b>	8486 2% / <b>13%</b>	212 0% / <b>0%</b>	20 0% / <b>0%</b>
Sbox6	144592 63%	9392 4% / <b>11%</b>	8515 4% / <b>10%</b>	10443 5% / <b>12%</b>	24581 11% / <b>29%</b>	21571 9% / <b>26%</b>	4473 2% / <b>5%</b>	3401 1% / <b>4%</b>	2152 1% / <b>3%</b>

Table 4. Fault vs error multiplicity on S-boxes output

## 7. Conclusions

This chapter presented a study on mechanisms involved for detecting faults-based attacks on crypto-processors. Using the example of a standard in symmetric cryptographic, fault-based attacks were discussed with respect to their requirements in terms of error multiplicity (spatial and timing characteristics). We presented countermeasures to fault-based attacks that consist in detecting errors on the ciphered information. We analyzed more precisely some error detection schemes based on code-redundancy, with respect to their cost and ability to detect errors occurring at run time. Analysis on error detection has been conducted according to the error multiplicity in order to check the ability of the protection schemes to detect exploitable errors. We also analyzed error detection schemes performances in terms of transient fault detection (natural or maliciously injected).

Current attacks using laser-based fault injection require errors localized on very few bytes, without need of large precision as for the attack launch instant. Error detection schemes can be used for preventing these attacks or detecting natural transient faults. Dedicated parity and CRC -based solutions exploit typical features of the cipher for optimization of cost factors (area, latency) and improvement of their error detection capacity. The correlation between the transient faults affecting the combinational parts of the circuit and the errors produced on sub-function outputs strongly depends on the implementation of the cipher. Experiments show that for some particular implementations, most of the faults results in only one or two erroneous output bits, while there is no internal faults affecting all the output bits. Such analysis on ciphering operations can justify simple and non expensive code-redundancy solutions.

## 8. References

- Bertoni, G.; Breveglieri, L.; Koren I.; Maistri, P. & Piuri, V. (2003), Error Analysis and Detection Procedures for a Hardware Implementation of the Advanced Encryption Standard, *IEEE Trans. on Computers*, Vol. 52., No.4, April 2003
- Blömer J. & Krummel, V. (2006), *Fault Based Collision Attacks on AES*, *Proceedings of FDTC 2006*, pp. 106-120.
- Blömer, J. & Seifert, J.P. (2003), Fault Based Cryptanalysis of the Advanced Encryption Standard (AES), *Proceedings of CHES 2003*, pp 162-181.
- Chen, C.N. & Yen, S.M. (2003), Differential Fault Analysis on AES Key Schedule and Some Countermeasures, *Proceedings of Australasian Conference on Information Security and Privacy 2003*, LNCS 2727, Springer-Verlag, pp. 118-129.
- Di Natale, G.; Flottes M.L. & Rouzeyre, B. (2007 B), An On-Line Fault Detection Scheme for SBoxes in Secure Circuits, *Proceedings of 13th IEEE International On-Line Testing Symposium, IOLTS 2007*, pp. 57-62.
- Di Natale, G.; Flottes, M.L. & Rouzeyre, B. (2007 A), A Novel Parity Bit Scheme for SBox in AES Circuits, *Proceedings of IEEE Design and Diagnostics of Electronic Circuits and Systems, DDECS 2007*, April 2007, pp. 1-5, DOI 10.1109/DDECS.2007.4295295
- Dusart, P.; Letourneux, G. & Vivolo, O. (2003), Differential Fault Analysis on A.E.S., *Applied Cryptography and Network Security*, Springer Ed., Vol. 2846/2003, pp 293-306.
- Eurosmart (2007), Vision paper 2020,  
<http://www.eurosmart.com/index.php/publications/vision-paper-2020.html>
- FIPS-197 (2001), Advanced Encryption Standard (AES), *Federal Information Processing Standards Publication 197*, <http://csrc.nist.gov/publications/>, November 26, 2001
- Giraud, C. (2005), DFA on AES, *Proceedings of 4th International Conference on AES 2005*, Springer publisher, pp 27-41.
- Karri, R.; Wu, K.; Mishra, P. & Kim, Y. (2002), Concurrent Error Detection Schemes for Fault-Based Side-Channel Cryptanalysis of Symmetric Block Ciphers, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 21, N. 12, December 2002, pp. 1509-1517
- Kim, C.H. & Quisquater, J.-J. (2007), Faults, Injection Methods, and Fault Attacks, *IEEE Design & Test of Computers*, Nov.-Dec. 2007, Vol.24, Issue 6, pp.544-545.
- Leveugle, R. (2007), Early Analysis of Fault-Based Attack Effects in Secure Circuits, *IEEE Transactions on Computers*, Vol. 56, N. 10, October 2007, pp. 1431-1434
- Maistri, P.; Vanhauwaert, P. & Leveugle, R. (2007), A Novel Double-Data-Rate AES Architecture Resistant against Fault Injection, *Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2007, DOI 10.1109/FDTC.2007.8, pp. 54-61
- Monnet, Y.; Renaudin, M. & Leveugle, R. (2006), Designing Resistant Circuits against Malicious Faults Injection Using Asynchronous Logic, *IEEE Transactions on Computers*, Vol. 55, N. 9, September 2006, pp. 1104-1115
- Moradi, A.; Manzuri Shalmani, M.T. & Salmasizadeh, M. (2006), A Generalized Method of Differential Fault Attack Against AES Cryptosystem, *Proceedings of Cryptographic Hardware and Embedded Systems*, CHES 2006, pp 91-100.
- Piret, G. & Quisquater, J.J. (2003), A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD, *Proceedings of Chess 2003*, pp 77-88.

- Reed, R.A.; Kinnison, J.; Pickel, J.C.; Buchner, S.; Marshall, P.W.; Kniffin, S. & LaBel, K.A. (2003), Single-event effects ground testing and on-orbit rate prediction methods: the past, present, and future, *IEEE Transactions on Nuclear Science*, Vol. 50, pp. 622-634, June 2003
- Wolkerstorfer, J.; Oswald, E. & Lamberge, M. (2002), An ASIC Implementation of the AES SBoxes, *Proceedings of The Cryptographer's Track at the RSA Conference on Topics in Cryptology 2002, Lecture Notes In Computer Science*, Vol. 2271; pp. 67-78
- Wu, K; Karri, R.; Kuznetsov, G. & Goessel, M. (2004), Low Cost Concurrent Error Detection for the Advanced Encryption Standard, *Proceedings of IEEE International Test Conference, 2004*. pp 1242- 1248
- Yen, C.H. & Wu, B.F. (2006), Simple Error Detection Methods for Hardware Implementation of Advanced Encryption Standard, *IEEE Trans. on Computers*, June 2006, Vol. 55, No.6, pp 720-731
- Zhang, X. & Parhi, K.K. (2002), Implementation approaches for the Advanced Encryption Standard algorithm, *IEEE Circuits and Systems Magazine*, Volume 2, Issue 4, Fourth Quarter 2002, pp. 24-46, DOI 10.1109/MCAS.2002.1173133



## **Fault Detection**

Edited by Wei Zhang

ISBN 978-953-307-037-7

Hard cover, 504 pages

**Publisher** InTech

**Published online** 01, March, 2010

**Published in print edition** March, 2010

In this book, a number of innovative fault diagnosis algorithms in recently years are introduced. These methods can detect failures of various types of system effectively, and with a relatively high significance.

### **How to reference**

In order to correctly reference this scholarly work, feel free to copy and paste the following:

K. Bousselam, G. Di Natale, M.-L. Flottes and B. Rouzeyre (2010). Fault Detection in Crypto-Devices, Fault Detection, Wei Zhang (Ed.), ISBN: 978-953-307-037-7, InTech, Available from:  
<http://www.intechopen.com/books/fault-detection/fault-detection-in-crypto-devices>

# **INTECH**

open science | open minds

### **InTech Europe**

University Campus STeP Ri  
Slavka Krautzeka 83/A  
51000 Rijeka, Croatia  
Phone: +385 (51) 770 447  
Fax: +385 (51) 686 166  
[www.intechopen.com](http://www.intechopen.com)

### **InTech China**

Unit 405, Office Block, Hotel Equatorial Shanghai  
No.65, Yan An Road (West), Shanghai, 200040, China  
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元  
Phone: +86-21-62489820  
Fax: +86-21-62489821