Error Analysis and Detection Procedures for a Hardware Implementation of the Advanced Encryption Standard

Guido Bertoni, Luca Breveglieri, Israel Koren, *Fellow*, *IEEE*,
Paolo Maistri, and Vincenzo Piuri, *Fellow*, *IEEE*

Abstract—The goal of the Advanced Encryption Standard (AES) is to achieve secure communication. The use of AES does not, however, guarantee reliable communication. Prior work has shown that even a single transient error occurring during the AES encryption (or decryption) process will very likely result in a large number of errors in the encrypted/decrypted data. Such faults must be detected before sending to avoid the transmission and use of erroneous data. Concurrent fault detection is important not only to protect the encryption/decryption process from random faults. It will also protect the encryption/decryption circuitry from an attacker who may maliciously inject faults in order to find the encryption secret key. In this paper, we first describe some studies of the effects that faults may have on a hardware implementation of AES by analyzing the propagation of such faults to the outputs. We then present two fault detection schemes: The first is a redundancy-based scheme while the second uses an error detecting code. The latter is a novel scheme which leads to very efficient and high coverage fault detection. Finally, the hardware costs and detection latencies of both schemes are estimated.

Index Terms—Advanced Encryption Standard, AES, fault tolerance, fault detection, parity codes.

1 Introduction

The Rijndael Advanced Encryption Standard (AES) algorithm is a secret-key crypto-system recently approved as standard by NIST [5]. AES is intended to replace the widely used DES and Triple-DES crypto-systems due to the last two's limited level of security [4], [6]. AES is an evolution of DES and extends it with respect to three different sets of features: the mathematical structure—AES is more complex than DES, requiring a larger number and more powerful basic operations; the control-path—AES uses longer keys than DES does; and the data-path—AES operates on larger blocks of data than DES.

Implementations of DES, both in software and in hardware, have existed for quite some time [14], [15]. Several software and hardware implementations of AES have recently been proposed. The software solutions have targeted various platforms [5], [12] with the goal of reducing the number of clock cycles required to encrypt a data block. Hardware solutions have been presented for field-programmable VLSI devices (e.g., FPGA implementations [7]). The objectives there were to increase the throughput while reducing the number of gates in the FPGA and to obtain reconfigurable devices able to cope with the different sizes allowed by AES for the keys and the data blocks. Custom devices, in contrast, are less flexible, but are more resistant against tampering or physical alteration than field-programmable ones. Some macrocell and coprocessor cryptographic architectures (also known as crypto-processors) of this kind have been proposed and evaluated at the simulation level [7], [8], [9], [10]. These crypto-processor architectures are particularly optimized for embedded systems and even smart-card systems.

Fault detection and possibly fault tolerance are undoubtedly key issues when designing a crypto-processor custom VLSI architecture for implementing the AES crypto-system since it is considerably more complex than the DES crypto-system it replaces. In fact, AES executes a very nonlinear algorithm and has an iterative structure requiring several repetitions of the same basic pattern of operations. Therefore, an AES crypto-processor is larger, more complex, and, hence, more likely to be subject to faults than the existing and commercially available DES crypto-processors [8], [9], [12]. Moreover, fault detection is a desirable property for preventing malicious attacks, aimed at extracting sensitive information, like the secret key, from the device [6], [11].

The issue of fault detection and tolerance in AES seems to be a new and mostly unexplored field. Karri et al. have recently addressed this topic in [11] from the perspective of preventing attacks based on malicious injection of faults. Their assumption is that, by suitably tampering with the device and analyzing the obtained erroneous outputs, sensitive data could be inferred. The proposed solutions consist of using various forms of redundancy to obtain an

- *G. Bertoni, L. Breveglieri, and P. Maistri are with the Department of Electronics and Information Technology, Politecnico di Milano, Piazza Leonardo Da Vinci n. 32, I-20133 Milano, Italy. E-mail: {bertoni, brevegli, maistri}@elet.polimi.it.*
- *I. Koren is with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003. E-mail: koren@ecs.umass.edu.*
- *V. Piuri is with the Department of Information Technologies, Advanced Research Center on Evolutionary Knowledge for Design Innovation by High-Performance Computing, University of Milan, Via Bramants 65, I-26013 Crema (CR), Italy. E-mail: piuri@elet.polimi.it.*

Manuscript received 14 June 2002; revised 30 Nov. 2002; accepted 30 Nov. 2002.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 117870.

attack-resistant architecture. These solutions have different detection time latencies and hardware costs and, in general, exhibit a large cost close to that of duplication.

However, the above work does not include either an analysis of the propagation of errors in AES or less expensive fault detection techniques, like those based on error detection codes. In [1], a preliminary study of the error propagation in AES was carried out but only for a simple fault model, namely, single transient faults. A simple fault detection technique was also introduced, based on the idea of exploiting the decryption block for checking the correctness of the encryption process. The analysis of the error propagation in AES has been extended in [2] to multiple and permanent faults and the presence of faults has been modeled at a finer-grained level. In both papers, the characterization of the behavior of AES in the presence of injected errors has been obtained by simulation.

In this paper, we present a comprehensive study of fault detection in a generic hardware VLSI implementation of AES. The paper summarizes previous experimental results concerning error propagation in AES for several different fault models, presents some theoretical interpretation thereof, and discusses previous and novel fault detection techniques. In particular, a simple but efficient error detection code for AES is developed and evaluated. This last result proves the nonobvious conclusion that it is practical to devise efficient error detection codes, even for a complex and nonhomogeneous algorithm like AES.

The paper is organized as follows: In Section 2, a brief overview of the AES algorithm is presented, including the implementation details which are necessary for understanding our proposed error detection schemes. Section 3 describes the analysis of error propagation in the encryption and decryption units for a simple single bit transient fault model [1]. The error analysis is then extended in Section 4 to a more practical fault model which also includes multiple faults and permanent faults [2]. This analysis allows us to obtain a rather comprehensive picture of the general behavior of AES in the presence of faults. The analysis is carried out by simulation since the structure of AES is too complex for an exhaustive theoretical analysis. Section 5 describes two fault detection algorithms. The first is a redundancy-based technique which has already been partially described in [1], while the second is novel and is based on exploiting error detecting codes, properly organized so as to fit AES (some hints are in [3]). Finally, Section 6 concludes the paper. Appendices A and B outline several mathematical proofs for the error detection codes which are proposed in Section 5. Appendix C outlines the proof of the coverage of the parity code scheme. Appendix D contains the evaluation of the hardware overhead due to such a parity code.

2 The Rijndael Algorithm

The Rijndael AES is a secret-key (symmetric) block cipher crypto-system [5] which encrypts (or decrypts) one block of data at a time. The encryption algorithm accepts one data block (or plain text) and the key and produces the encrypted data block (the input and output data blocks are of identical size). The decryption algorithm accepts one

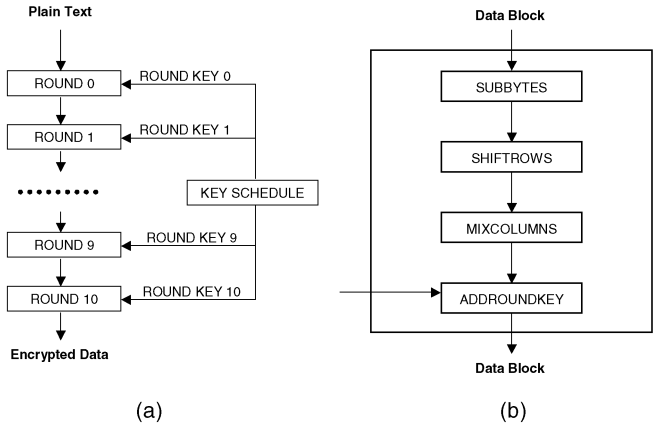


Fig. 1. (a) The data-path for data block and key size of 128 bits, (b) generic structure of one internal round.

encrypted data block and the key and outputs the plain text. Both encryption and decryption use the same secret key.

Internally, the AES encryption algorithm can be partitioned into two processes, performed in parallel: encryption and key schedule. In the case where the AES encryption process is executed by a dedicated device (or crypto-processor), these two processes can be viewed as the data-path and the control-path of the complete AES crypto-processor. The decryption algorithm is similarly partitioned into the decryption and inverse key schedule processes. Encryption and decryption are mathematically inverse, as are key schedule and inverse key schedule.

2.1 The Data-Path

AES is a flexible crypto-system allowing the sizes of the data block and the secret key to be any combination of 128, 196, and 256 bits. However, NIST has restricted the size of the data blocks to only 128 bits, while the key still has all three options. The version with data block and key of equal size of 128 bits each is regarded as the basic and most practical one and has an adequate security level for most civil applications.

AES has an iterative structure consisting of a repetition of a round which is applied to the data block to be encrypted for a fixed number of times. The number of rounds is determined by each key size. For the three key sizes of 128, 196, and 256 bits, a number of 10, 12, and 14 rounds is required, respectively, plus an initial special round (called round 0). Fig. 1a shows the steps of the Encryption process for a 128-bit key.

A round consists of a fixed sequence of transformations. Except for the first round (round 0) and the last round, the other rounds (internal rounds) are identical and consist of four transformations each. The first and last rounds are incomplete. The four round transformations are called SubBytes, ShiftRows, MixColumns, and AddRoundKey, see Fig. 1b. The transformation AddRoundKey is the point where the secret key enters the Encryption process and contributes to the final result.

The four round transformations are invertible, hence the round itself is invertible. The inverse round consists of the sequence, in reversed order, of the inverses of the four transformations.

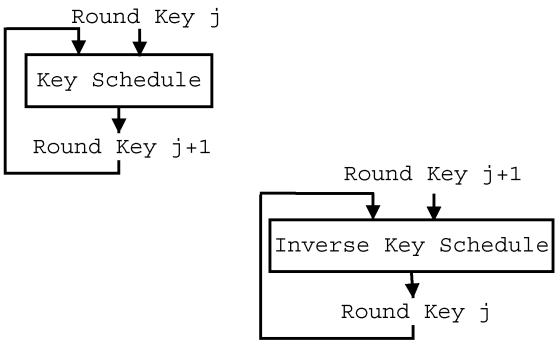


Fig. 2. The control-path for a key size of 128 bits.

2.2 The Control-Path

Each round accepts a (partially processed) data block and a round key and outputs a (further processed) data block. All round keys have the same size as the secret key, but there is a distinct round key for each round. The round keys are ultimately derived from the secret key by means of the key schedule process. More precisely, if sk is the secret key and rk_j is the jth round key (for $j \geq 1$), then key schedule computes $rk_{j+1} = KS_j(rk_j)$ as a function of the previous round key rk_j; the process starts with $rk_0 = sk$. Key schedule is invertible: Fig. 2 shows the inputs and outputs of the key schedule and inverse key schedule processes. The reader is referred to [5] for further details.

The concatenation of the secret key and of all the round keys is a sequence of bits called *key material*. Basically, there are two methods for calculating the round keys, called *key unrolling* and *key on-the-fly* [13]. The former method computes and stores the key material in advance, accessing it whenever a round key is required. The latter computes each round key just before starting the related round and discards it immediately after completing that round.

2.3 Basic AES Operations

All four internal transformations of one AES round work on byte elements, and are rooted in the algebra of finite fields (Galois fields, GF) [16]. The finite fields of interest for AES are the binary fields, of type $GF(2^n)$. The integer n ($n \geq 1$) identifies the number of bits used to represent the field elements. The field $GF(2) = \langle \{0,1\}, +, \cdot \rangle$ is the well-known field of addition ($+$) and multiplication ($\cdot$) modulus 2; these two operations are executed by the XOR and AND logic gates, respectively.

The basic operations of AES are defined over elements of the field $GF(2^8)$, i.e., on byte elements of $n = 8$ bits each. AES uses the standard basis, or polynomial, representation for the field $GF(2^8)$ [16]. One byte can be represented as a polynomial $A(x)$ of degree 7 or less, with coefficients over the field $GF(2)$:

$$A(x) = \sum_{i=0}^{7} a_i x^i = a_0 + a_1 x + \cdots + a_7 x^7, \tag{1}$$

where $a_i \in \{0,1\}$ for every $0 \leq i \leq 7$. For convenience, a byte can also be represented in binary or hexadecimal in addition to its polynomial presentation. For instance, the binary number 0010 1101, or in hexadecimal $2d$, represents the polynomial $x^5 + x^3 + x^2 + 1$. AES uses the following irreducible polynomial $\Phi(x)$ of degree 8 as generator for the finite field $GF(2^8)$:

$$\Phi(x) = x^8 + x^4 + x^3 + x + 1. \tag{2}$$

The round transformations use the following basic operations over polynomials:

$$\begin{aligned} A(x) \oplus B(x) &= A(x) + B(x) \bmod \Phi(x) \\ A(x) \otimes B(x) &= A(x)B(x) \bmod \Phi(x) \\ A^{-1}(x) &= B(x) \quad \text{s.t.} \quad A(x) \otimes B(x) = 1. \end{aligned}$$

Addition $\oplus$ reduces to a simple bit-wise XOR of the coefficients of the two summand polynomials, thus not requiring the use of the generator polynomial $\Phi(x)$. Multiplication $\otimes$ and inversion $()^{-1}$, in contrast, do require the use of the generator $\Phi(x)$. Several algorithms for computing finite field arithmetic operations can be found in [16].

2.4 Round Transformations

A precise mathematical formulation of the four round transformations is presented in what follows. This provides the necessary background for the presentation of the error detection schemes in Section 5. Consider, for simplicity, the case of a data block and a secret key having the same size of 128 bits. The data block db is partitioned into 16 bytes db_i, with $0 \leq i \leq 15$. This byte sequence is rearranged as a 4×4 matrix S, called "state matrix" (or simply state).

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

Denote by $s_{r,c}$ (with $0 \leq r, c \leq 3$) the element in row r and column c of the state S, then the rearrangement scheme is: $s_{r,c} = db_{r+4c}$. In other words, the state S is organized by columns. Each byte $s_{r,c}$ is an element of the finite field $GF(2^8)$. The state S is modified by the round transformations as follows.

SubBytes. Every element $s_{r,c}$ of the state S is first inverted and then processed through an affine transformation T:

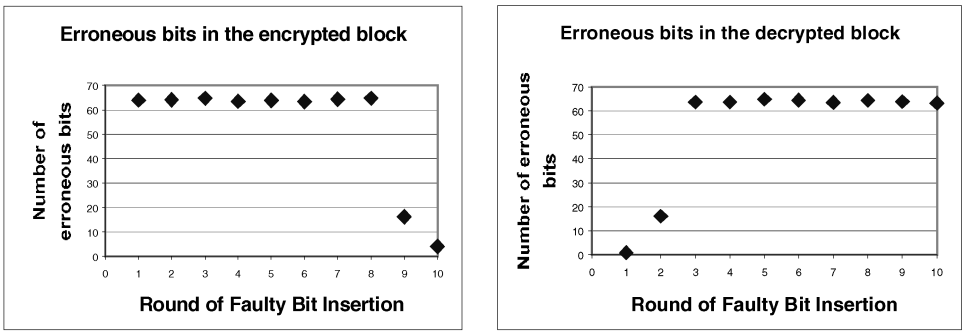
$$s_{r,c} \mapsto T(s_{r,c}^{-1}) \quad \text{for} \quad 0 \leq r, c \leq 3. \tag{4}$$

Clearly, SubBytes is a nonlinear transformation, mainly due to the inversion it contains.

The SubBytes transformation operates independently on each byte of the state S; therefore, it can be computed in parallel for all the state elements. For reasons of efficiency, in most practical implementations of AES, SubBytes is computed in advance and stored in a look-up table of $2^8 = 256$ elements. In this paper, it is assumed that SubBytes is implemented as such a look-up table, which is referred to as the *Sbox*.

ShiftRows. The rows of the state S are progressively rotated, as follows:

$$S \mapsto \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,1} & s_{1,2} & s_{1,3} & s_{1,0} \\ s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3,3} & s_{3,0} & s_{3,1} & s_{3,2} \end{bmatrix}. \tag{5}$$



(a) (b)

Fig. 3. Mean number of erroneous bits in the encrypted (a) and the decrypted (b) data block, versus the injection round of the faulty bit.

The first row is left unchanged, the second row is rotated one byte position to the left, the third row two byte positions, and the fourth row three. ShiftRows is a linear transformation.

MixColumns. This is a linear transformation operating on the elements of the state S as shown below, where $\alpha = 02 = x$ and $\beta = 03 = x + 1$ are fixed coefficients.

$$s_{0,c} \mapsto \alpha \otimes s_{0,c} \oplus \beta \otimes s_{1,c} \oplus s_{2,c} \oplus s_{3,c}, \quad (6)$$

$$s_{1,c} \mapsto s_{0,c} \oplus \alpha \otimes s_{1,c} \oplus \beta \otimes s_{2,c} \oplus s_{3,c}, \quad (7)$$

$$s_{2,c} \mapsto s_{0,c} \oplus s_{1,c} \oplus \alpha \otimes s_{2,c} \oplus \beta \otimes s_{3,c}, \quad (8)$$

$$s_{3,c} \mapsto \beta \otimes s_{0,c} \oplus s_{1,c} \oplus s_{2,c} \oplus \alpha \otimes s_{3,c}, \quad (9)$$

for every column $0 \leq c \leq 3$. Each transformed byte is a linear combination of the four bytes of the state S in the same column c.

AddRoundKey. Every 128-bit round key rk is partitioned into 16 bytes rk_i, $0 \leq i \leq 15$, similarly to the data block. AddRoundKey is a linear transformation and modifies the elements of the state S as shown below.

$$s_{r,c} \mapsto s_{r,c} \ \oplus \ rk_{r+4c} \quad \text{for } 0 \leq r, c \leq 3. \quad (10)$$

This is equivalent to rearranging the round key rk as a matrix, similarly to the data block db and adding the two matrices. It is also equivalent to bit-wise XORing the two 128-bit words db and rk.

The above presentation of the four round transformations has been tuned to a data block and a key of size 128 bits each. The transformations can be generalized to the other allowed sizes by using rectangular state matrices with more columns. The reader is referred to [5] for details.

The Key Schedule functions KS_j process the round keys using the same basic constituents of the round transformations. A function KS_j is a combination of the following operations: a right rotation, the SubBytes operation, and an addition of a byte constant.

3 Error Analysis: Single Fault

In this section, the error propagation behavior of the data-path (i.e., the encryption or decryption process) is studied. The purpose of this study is to understand the effect of a fault occurring during the execution of the algorithm on the final result. This is an important first step when developing fault detection and tolerance schemes. For simplicity, the single faulty bit model is adopted in this section, i.e., only a single bit may become faulty at any given time instant. Furthermore, since the encryption and decryption algorithms include a large number of steps, attention is restricted to single faulty bits inserted at the beginning of each round rather than during the intermediate steps within a round.

3.1 Error Propagation in the Data-Path

Fig. 3 shows the results of simulation experiments in which a faulty bit has been injected into the AES data-path. A data block and key both of size 128 bits were used in most of the experiments, but it has been verified that the observed behavior is similar for the other two admissible key sizes. In these simulations, attention has been focused on the effect of an injected fault on the encrypted result and on the result of the decryption, where "effect" means the number of erroneous bits caused by a single faulty bit injected at some stage of the computation.

From Fig. 3a, it is possible to see that a faulty bit inserted in the first round of encryption causes a large number of erroneous bits in the final encrypted data. Applying decryption to the corrupt data reconstructs a decrypted block containing a single faulty bit. This behavior should be expected since the AES algorithm is invertible. Still, injecting a single error in the input message in any round between 2 and 8 yields a corrupt encrypted message which is considerably different from the correct one. Our simulations have shown that, on the average, 64 output bits were erroneous. Note, however, that if the faulty bit is inserted in the last two rounds of encryption, it spreads over a much smaller number of bits in the final enciphered message (1 or 16 versus 64 in earlier rounds). Similarly, injecting a single faulty bit in the early rounds of decryption yields a decrypted message which is quite different from the original correct message, as shown in Fig. 3b. No faults were injected prior to round 0 because this would be equivalent to considering a different message.

3.2 Error Propagation in the Control-Path

Another part of the algorithm implementation that can be affected by faults is the key schedule. A complete key unrolling is subject to two types of errors: either a single faulty bit corrupting the stored key material or a faulty bit injected during the round key computation process, spreading to many bits. In contrast, the key on-the-fly approach can be subject only to the second type of error since the key material is never completely computed and stored. A faulty bit injected during the unrolling process may cause a large number of erroneous bits in the next round keys.

Both situations have been simulated: The former case is equivalent to the injection of a single error in the data-path since the round key is added to the state matrix at the end of each round. As for the latter case, [1] and [2] show that the number of erroneous bits obtained in the key material can be as high as 360 out of 1,408 bits composing the complete key material.

3.3 The Effect of an Error in the Control-Path on the Data-Path

When the data-path is assumed to be fault-free and the key scheduling is affected by the injection of a single faulty bit at some round, it has been verified that a faulty bit injected in the early rounds causes a high number of erroneous bits in the decryption process. If the erroneous round key is used for decryption, it is not possible to detect the presence of a faulty bit in the key material. The sender will be unable to realize that the transmitted encrypted data is corrupted and the receiver will decrypt useless data. Consequently, special attention must be paid to the fault management of the round key.

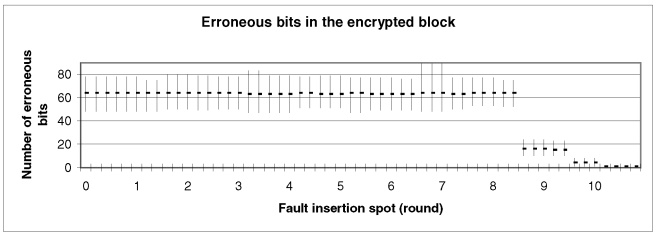
4 Error Analysis: Internal and Multiple Faults

In this section, we extend the fault model to first include single faults during the internal transformations of a round, and then to multiple faults.

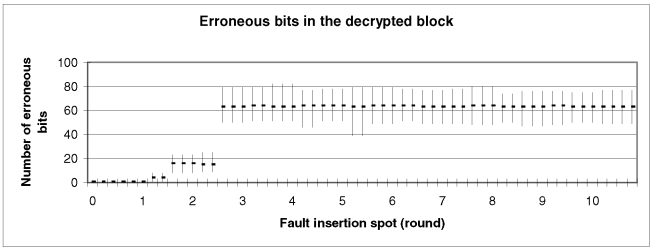
4.1 Internal Faults

First, the effect of a single fault at any step of the process is analyzed. A fault injected during the very first round (round 0) is comparable to encoding a different input. The only operation performed at this stage is the key addition, which does not interfere with the error propagation: This is confirmed by Fig. 4, where it is shown that the decoded output differs from the correct one by exactly one bit.

The injection of a fault during one of the inner rounds is more complicated and it is necessary to follow the errors as they propagate along the execution path. The generic Encryption round of AES consists of four round transformations: SubBytes, ShiftRows, MixColumns, and AddRoundKey (see Section 2.4). Hence, there are five positions where an error may be injected, from the beginning (before any other operation) to the end of the round. However, a fault injected at the end of a round, provided that it is not the last round, is equivalent to an injection at the beginning of the following round. Thus, one position of the possible five need not be analyzed.



(a)



(b)

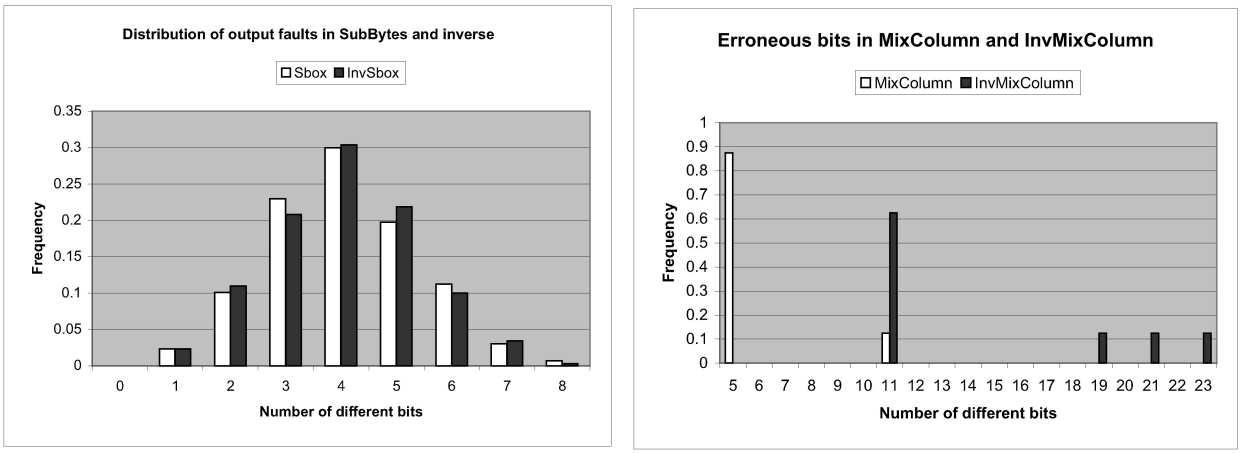
Fig. 4. Effects of a transient fault in the state for encryption (a) and decryption (b).

The propagation of a single fault injected in the other four positions is influenced by the execution of the round components. The result can be classified into two cases: The fault spreads considerably or it affects only one output of the component. The latter situation includes the ShiftRows and the AddRoundKey transformations, where the error is only moved within a row of the state matrix or left untouched, respectively. The relative position of the fault with respect to these transformations does not affect the amount of errors in the output since only trivial operations like byte rotation or linear ones like XOR are involved. The two remaining round transformations are more complex and will therefore greatly influence the propagation of errors.

Fig. 4 shows how the number of erroneous bits changes as the position of fault injection is changed from round to round and within each round, using a specific input and injecting a fault into every single bit. The differences are more apparent in the boundary rounds, where their effects are not masked. Examining the first and the last rounds confirms that the positions most sensitive to faults precede the execution of Sbox and their respective inverse transformations.

An analysis of the way these two round transformations spread a single fault leads to some interesting observations. The application of the Sbox substitution creates a number of errors ranging from 1 to 8 with the most common being 4, as Fig. 5a shows. Such data suggest that the number of erroneous output bits follows a binomial distribution, implying that the result would actually be random. Applying the χ^2 test to the frequencies generated by the simulation shows that the data fits the suggested model very well. Further analysis focusing on single output bits has shown that the distribution of the fault is quite uniform, that is, every bit is equally likely to be erroneous.

The other complex round transformation is MixColumns: The distribution of the number of errors generated from a single injected fault by this transformation is completely discrete. When injecting a fault before MixColumns, either 5



(a) (b)

Fig. 5. Effect of the injection of a single faulty bit on the outputs of Sbox and InvSbox (a) and on the outputs of MixColumns and InvMixColumns (b).

or 11 errors are obtained at the output; similarly, when injecting a fault in the InvMixColumns, the number of errors is 11, 19, 21, or 23; both cases are shown in Fig. 5b. This behavior is due to the finite field multiplication performed in (Inv)MixColumns. In fact, MixColumns shows the same fault distribution pattern as the finite field product in $GF(2^8)$, although scaled in the number of bits involved. Consider, for instance, Fig. 6: It shows the effect of the injection of a fault into the operation $A(x) \otimes x$, with $A(x) \in GF(2^8)$. When the fault is injected into the most significant bit of $A(x)$, it is spread by the field generator polynomial $\Phi(x)$ over a large number of bits, while, for any other injection position, the error is only shifted. A similar error pattern has been identified in the spreading of errors caused by a single faulty bit injected into one column of the state, when performing MixColumns and InvMixColumns [2].

4.2 Multiple Faults

In Section 3, we considered the case when a fault affects only one bit during the computation. In this section, we analyze the behavior of the AES implementation when multiple faults are present. Two such situations are studied: multiple temporary faults and permanent faults; the similarities and the differences with respect to the single fault model are presented.

A single error occurring in the inner part of encryption has led to completely different outputs, both encrypted and decrypted (see Fig. 4). The average number of different bits is about 64, which is the expected value of a completely random string since a random single bit is correct half the time. Injecting two independent faults at different rounds shows similar results, as depicted in Fig. 7. Only when the faults are injected at the very first or at the very last rounds are the outputs partially related to the correct value (about 20 or fewer erroneous bits), while, in most cases, the final output is random. A permanent fault sets the value of a specific bit to a constant 0 or 1 and may be the result of a short or open circuit. This yields a variable number of injected faults, depending on the original bit value: In the worst case, it may amount to one error in each round. The results of this experiment resemble the results of injecting multiple temporary faults and, as the number of temporary faults increases, the results of the two experiments get closer. A similar behavior has been observed when two or more faults were injected in the key material. Experiments with injecting multiple faults lead to two important observations. First, only a very few experiments yielded a small number of erroneous bits; in most cases, the number of erroneous bits was 64 on average, leaving an apparent gap between the common case and the few cases with 20 or less errors (see Fig. 7). Second, no masking effects of faults were revealed during our extensive experiments. A masking effect can still be obtained by injecting one fault into the state and a second one into the corresponding bit in the key material. However, such faults are an unlikely event. Faults

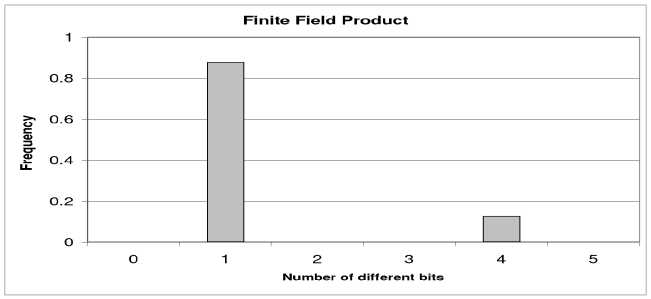


Fig. 6. Spreading of errors caused by the injection of a fault into the multiplication $A(x) \otimes x$.

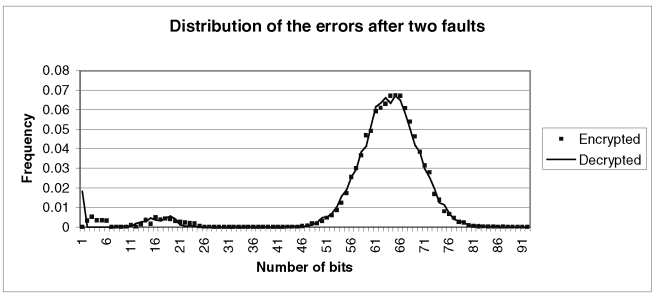


Fig. 7. Effect of two fault injections in the state for the encrypted and decrypted output.

injected at different rounds would not normally overlap due to the nonlinearity that is spreading the error very efficiently.

Injecting multiple faults during the MixColumns transformation gives different results as the injection affects the direct or the inverse transformation. While the latter quickly becomes stable and distributes the error uniformly over the output due to its complexity, the former is much simpler and keeps the same fault distribution pattern, even with multiple faults. The number of output errors can belong to a wider set of possible values, although it maintains some sort of parity property, i.e., the injection of an even (odd) number of faults results in an even (odd) number of output errors.

5 Fault Detection Techniques

A proposal for error detection in the data-path of AES was described in [11]. The goal there was to prevent an attacker from breaking the cipher system by injecting one or more incorrect bits. This paper has an additional objective: to detect a fault in order to prevent the transmission and use of incorrect data. This issue is important as any hardware implementation of AES is bound to be complex and, consequently, likely to be subject to fault occurrences. In this section, two techniques for fault detection in a generic hardware custom implementation of an AES crypto-processor are presented. The first technique is based on redundancy (see also [1], [2]) and is similar to those presented in [11]. The second one is a novel technique which is based on error detecting codes, namely, a suitably designed multiple parity bit code. The latter proves to be very efficient and has a rather low hardware overhead.

Our objective is to develop fault detection techniques which will be independent of the particular hardware implementation chosen. To this end, we make the following assumptions:

- The AES crypto-processor is partitioned into three basic hardware modules: encryption, decryption, and key schedule (an inverse key schedule module is a possible enhancement).
- All the modules have in common the same basic operations; hence, only the encryption module is examined here in detail since most conclusions will hold for the remaining modules as well.

In what follows, the two fault detection techniques are described and validated, their characteristics are evaluated through simulation, and their hardware costs are estimated.

5.1 Redundancy-Based Technique

The redundancy-based solution for implementing fault detection in the encryption module is based on the idea of performing a test decryption immediately after the encryption and then checking whether the original data block is obtained. If a decryption module is already present in the implementation, the hardware overhead reduces to the cost of a comparator for two data blocks of 128 bits. Otherwise, the overhead is close to 100 percent since the decryption module is very similar to the encryption one. The time penalty in either of these two cases is the time required to decrypt a data block, plus the time required for the comparison. Clearly, this technique is independent of the adopted fault model.

A finer-grained error detection, able to inspect the internal stages of each of the four transformations forming one round (see Section 2.4), would allow a smaller fault detection latency. This would prevent the execution of useless operations on already corrupt data, but would require a larger hardware overhead since comparison will have to be done at the transformation level. Considering the four transformations in a round, suitable points where such finer error detection would be recommended are the SubBytes and the MixColumns transformations, due to their evidently higher complexity compared to ShiftRows and AddRoundKey.

The key schedule module deserves particular attention since a fault in this part of the AES crypto-system is not detected by the techniques used for detecting faults in the data-path. In [1], it was suggested to use an inverse key schedule module,[1] allowing us to reconstruct the previous round key from the current one. If the result of the inverse key schedule module matches the round key computed by the key schedule module, the computation is correct and can proceed; otherwise, a fault has been detected. Clearly, this scheme is also independent of the selected fault model.

5.2 Error Detecting Codes

Error detecting codes (EDCs) have been widely used in practice. EDCs may at first seem unsuitable for implementing error detection in AES since AES is a rather non-homogeneous and strongly nonlinear algorithm and because errors spread quickly over the data block (see Sections 3 and 4). In this section, an efficient EDC scheme for AES will be described and evaluated. It achieves a high level of fault coverage at a limited hardware overhead cost and low detection latency.

5.2.1 The Basic Principle

One of the simplest EDCs is perhaps the well-known parity code, which is capable of detecting all single bit errors and those multiple bit errors where the number of errors is odd. We cannot, however, employ just a single parity bit for fault detection in the AES for the following reasons:

- As shown in Sections 3 and 4, errors spread quickly throughout the data block as encryption goes on and, on the average, about half of the state bits become corrupt. Hence, the fault coverage of the parity code would be at best around 50 percent, which is unacceptable in practice.
- Predicting the parity bit for the various round transformations is a complex and slow task due to the large size of the data block (128 bits): The parity bit would have a global dependence on all information bits.

To circumvent these problems, we propose associating one parity bit with each byte element of the state matrix S (see (3)), for a total of 16 parity bits. These parity bits can be

1. Normally, this module would not be necessary since the key material can be stored and read back in reverse order to feed the round keys to the decryption module.

arranged as a 4×4 parity matrix, the bit elements of which are in one-to-one correspondence to the byte elements of the state matrix S. Each parity bit is computed so that the parity of the data byte and the associated parity bit will be even.

The parity matrix certainly allows us to detect all single bit errors and all errors consisting of an odd number of erroneous bits. It can also detect some (possibly many) errors consisting of an even number of erroneous bits, provided that the erroneous bits are distributed over the state S in such a way that at least one byte of the state is affected by an odd number of erroneous bits. Moreover, each parity bit will now depend only on a limited portion of the data block, which may lead to a considerable reduction in the complexity of the parity prediction process.

To implement this coding scheme, it is necessary to develop, for each round transformation, a method for predicting the output parity, given the input state and the input parity. We then need to schedule checkpoints during the encryption process. At least one checkpoint is required, but possibly two or more for increasing the fault coverage and reducing the detection latency.

We next describe the structure of the parity bits' prediction and checking scheme, we then verify the characteristics of our scheme through simulation and estimate its cost. Further details about the coding scheme, its fault coverage, and hardware overhead are included in Appendices A through D.

5.2.2 Structure of the Coding Scheme

A parity prediction algorithm must be designed for each of the four round transformations employed by the encryption module. Since all byte elements of the output state for each transformation are computed in parallel, we must do the same with the output parity bits. We present in what follows our proposed parity prediction algorithms for the individual transformations.

SubBytes (or Sbox). The Sbox is usually implemented as a 256×8 bits memory, consisting of a data storage section and an address decoding circuit. The incoming data bytes will normally have properly generated even parity bits. To generate the outgoing parity bits, an even parity bit can be stored with each data byte in the Sbox memory, which will now be of size 256×9 bits. To detect input parity errors and some internal memory (data or decode) errors, we propose replacing the original 8-bit decoder with a 9-bit one, yielding a 512×9 memory. If a 9-bit address with an even parity is decoded, the corresponding output byte with its associated even parity bit is produced. Otherwise, a constant word of 9 bits with a deliberately odd parity is output, e.g., $00000000\,1$. Thus, half of the entries in the Sbox memory will be deliberately wrong.

There is still one type of internal memory error which will not be detected by the above scheme. These are faults in the address decode circuitry which may result in accessing a wrong location that has a valid even parity bit. If such faults are expected, we can add a separate 256×1-bit memory which will include the predicted parity bit for the correct output byte. This separate memory 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 bit to each location in the small memory, thus increasing its size. Comparing the output of this memory to the appropriate output bits of the main Sbox memory will allow the detection of most of the addressing circuitry faults.

ShiftRows. The prediction of the output parity bits is straightforward: It is simply a rotated version of the input parity bits following (5).

MixColumns. The prediction of the output parity bits of MixColumns is the most mathematically complex one. The detailed solution is described in Appendix A. The final set of equations for predicting the parity bits are, however, quite simple; see (15)-(18) in the appendix.

AddRoundKey. The prediction of the output parity bits is almost straightforward: It consists of adding the input parity matrix associated with the data block to the parity matrix associated with the current round key; see (10) for details.

The complete prediction scheme for one round is obtained by cascading the prediction schemes of the four round transformations. To check the parity bits and generate a parity error flag, we need a set of byte parity generators and comparators which will compare the predicted parity bits to the generated parity bits (see Appendix D for a detailed discussion of the required hardware).

It is also necessary to decide on the scheduling of the parity checks. Assuming that the rounds are computed sequentially, the three possible choices are:

1. Perform a check at the output of each round transformation. The resulting detection latency is the shortest possible, but four parity checkers are needed.
2. Perform a check only at the end of every round. The detection latency is longer, but only one parity checker is needed.
3. Perform a single check at the end of the last round. The detection latency is the highest and, as in case 2, only one parity checker is needed.

Each of these scheduling policies will somewhat slow down the encryption due to the parity check circuitry. Policy 1 is the most expensive in terms of extra encryption delay and hardware costs. However, this policy will yield the highest (among the three) fault coverage with the smallest detection latency. Policies 2 and 3 are less expensive, but have a higher latency and may have a lower fault coverage.

In the next section, we will show that even policies 2 and 3 reach a high fault coverage, namely, equal to 100 percent, for the single faulty bit model and the multiple faulty bit model of odd order (for single faults, the proof is in Appendix C). This is a nonobvious result since the behavior of AES, as described in Sections 3 and 4, is highly dispersive with respect to errors and this may, in principle, cause error masking.

A similar EDC scheme can also be adopted for the inverse rounds forming the Decryption module; see Appendix B for the prediction scheme for the inverse round transformation InvMixColumns, which is the most complex one of the four. The same approach (with a few

adjustments) also works for the (inverse) key schedule module as this module employs the same basic operations as the encryption (and decryption) module.

5.2.3 Fault Coverage of the Proposed Parity Code

In this section, we describe the results of extensive simulation experiments which were carried out to evaluate the fault coverage of the proposed parity-based EDC scheme for the encryption module. We start with single bit faults injected into the data block at the beginning of the rounds, i.e., faults are not injected between the round transformations. Six types of tests were performed with data block and key of 128 bits.

1. Five thousand data blocks were selected randomly and a single bit error was injected into every position of the data block at each of the 10 rounds. The total number of tests of this type has been $5,000 \times 10 \times 128 = 6.4 \times 10^6$. All these tests used the same secret key. Our parity bits scheme detected all the faults.
2. Five thousand secret keys were randomly selected and used with the same 128-bit data block. All possible single bit errors were injected, as in (1), for a total of 6.4×10^6 tests. Here too, all the faults were detected by our parity scheme.
3. One hundred random secret keys and 1,000 random data blocks were selected and every data block was encrypted with each secret key. One thousand two hundred eighty single bit errors were injected into every encryption for a total of 1.28×10^8 tests. All the faults were detected.

In the above three types of tests, the parity check was performed at the end of the 10th round. In the next type of tests, the parity check was instead done at the end of the single round performed.

4. Five hundred thoursand random data blocks were selected and a single bit error was injected in each position of the data block. A single round was then performed, yielding 100 percent fault coverage. The total number of tests of this type was $5 \times 10^5 \times 128 = 6.4 \times 10^7$.

The next two types of tests consider a single simplified round consisting only of SubBytes and MixColumns since these transformations affect the error propagation in the most complex way. The parity check is performed at the end of the (simplified) round. The observed fault coverage has been 100 percent.

5. Two hundred fifty-six 32-bit data words of the type $(x000)_8$ were considered and a single bit error was injected into the first byte (the one that is varying). The total number of tests of this type was $256 \times 8 = 2,048$. All the faults were detected.
6. One thousand 128-bit random data blocks were selected and a single bit error was injected in each position of the data block. The number of tests of this type was $1,000 \times 128 = 1.28 \times 10^4$. Again, all the injected faults were detected.

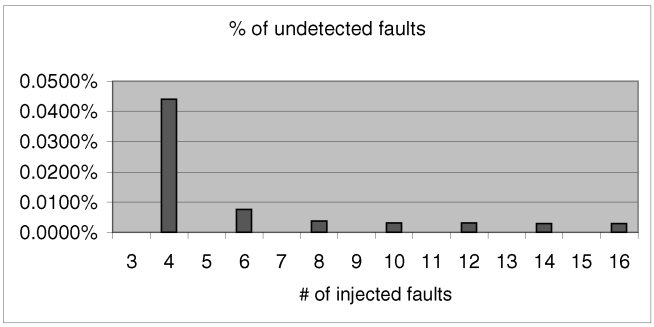


Fig. 8. Percentage of the undetected multiple faults injected in the encryption module.

These six types of tests strongly suggest that the parity-based EDC achieves a 100 percent fault coverage for single bit faults. In fact, it can be proven that: *The proposed parity-based EDC with a single checkpoint scheduled at the end of the last round is capable of detecting every single bit fault injected into the data block in the encryption module, at the beginning of the rounds or between two round transformations*. The proof is in Appendix C; the appendix states clearly the assumptions regarding the fault model and the scheduling of checkpoints.

It remains to investigate the detection capabilities of the parity-based EDC in the presence of multiple bit faults. An experiment has been carried out, injecting multiple bit faults (between 2 to 16) at the beginning of the rounds in the encryption module, with randomly selected data block and secret key. 10^7 encryptions have been simulated for every number of faults from 2 to 16. Fig. 8 shows the percentages of undetected faults, for 3 to 16 injected faults.

For double faults, the observed percentage of undetected faults was 0.875 percent, but it is not shown in Fig. 8 to avoid flattening to 0 of all the remaining percentages. One notable result is that all odd-order faults (i.e., multiple faults of order 3, 5, etc.) were always detected. The percentage of undetected even-order faults drops slowly to about 0.003 percent and remains stable at this value up to faults of order 100 and over, with a very small deviation.

Further analysis of the simulation results has revealed that the relatively high percentage of undetected double faults is mostly due to injections of both bit errors into the same data byte; an event which clearly causes masking. The probability of injecting all faults of an even order into the same data byte[2] decreases with the order of the fault. This explains why the percentages in Fig. 8 are decreasing. Due to the high dispersion of errors in AES (see Sections 3 and 4), it is reasonable to expect that such behavior remains essentially unchanged when the faults are injected between two round transformations.

Fig. 8 shows that the detection of odd-order faults reaches 100 percent. It can be proven that: *The proposed parity-based EDC with a single checkpoint scheduled at the end of the last round is capable of detecting every multiple fault of odd order, where the bit errors are injected into the data block at the beginning of the rounds or between two round transformations*. The proof is omitted for the sake of brevity.

2. Or also into two or more data bytes, but distributed in subgroups of even order each.

In summary, the coverage of the parity-based EDC is very high and, for single bit faults and multiple bits faults of odd order, it reaches 100 percent. For multiple faults of even order, the coverage is below 100 percent, but very close to it. Asymptotically, the global coverage converges to 99.997 percent.

5.2.4 Cost of the Proposed Parity Code

The cost in terms of hardware overhead for the parity-based EDC described above is limited. In Appendix D, it is shown that this overhead falls in the range 10-20 percent with respect to the nominal hardware cost of the encryption module, for the two checking policies 2 and 3 described in Section 5.2.2. Such a cost is acceptable since the resulting fault coverage is high and is comparable to the cost of other current fault detection circuits, like those used in memories. The detection latency is relatively short, for checking policy 2, or longer, for checking policy 3. Since the fault coverage is approximately the same for both policies, they offer two feasible solutions and the choice depends on the time constraints of the application.

These fault coverage and cost figures are likely to extend —with only a little modification—to the other modules forming an AES crypto-processor: decryption and (inverse) key schedule. The reasons behind this conjecture have already been discussed in Sections 2 and 5.2 and rely mainly on the fact that all the AES modules have the same, or very similar, basic operations in common. Therefore, the proposed parity EDC scheme is an efficient and low-cost fault detection technique for AES.

6 Conclusions

A detailed analysis of the behavior of the AES crypto-system in the presence of faults has been carried out. This analysis summarizes and integrates those presented in [1], [2]. Previous studies (e.g., [11]) have only considered the data-path, ignoring the key schedule. The behavior of AES in the presence of faults seems to be almost independent of the assumed fault model and is highly dispersive.

Two proposals for fault detection have been presented in this paper. The second one, which is based on the use of parity codes, exhibits very good fault coverage, limited hardware overhead cost, and short detection latency. For single bit faults and multiple bit faults of odd order, it has been proven that (under reasonable assumptions) the fault coverage of the parity-based detection technique is 100 percent. Future research directions include a wider exploration of the application of parity-based EDCs to AES, as well as the exploration of fault tolerance techniques, based on error correcting codes.

Appendix A

Parity Prediction in MixColumns

In this appendix, we describe the formal construction of the parity prediction scheme for the MixColumns round transformation (see Section 2.4).

The parity operator $p()$ is a function of the type:

$$p : GF(2^n) \rightarrow GF(2),$$

calculated as

$$\sum_{i=0}^{n-1} a_i x^i \mapsto \sum_{i=0}^{n-1} a_i, \qquad \text{for any } n \geq 1.$$

In other words, the coefficients a_i of the polynomial $A(x)$ are added modulus 2. It is also well-known that $p(A(x)) = A(x) \bmod (x+1)$. Clearly, the parity is a linear operator, i.e., $p(A(x) \oplus B(x)) = p(A(x)) + p(B(x))$. If $p(A(x)) = 0, 1$, then the polynomial $A(x)$ is said to have an even or odd parity, respectively.

The following lemma provides the basis for predicting the parity bits for the outputs of the MixColumns transformation, assuming that the inputs and their corresponding parity bits are known.

Lemma A.1 (Parity of product). *Given the following two polynomials with coefficients over $GF(2)$, for some $n \geq 1$,*

$$A(x) = \sum_{i=0}^{n-1} a_i x^i \text{ and } G(x) = x^n + \sum_{i=0}^{n-1} g_i x^i,$$

where $a_i, g_i \in \{0,1\}$, denote, by $p_A = p(A(x))$ and $p_G = p(G(x))$, their respective parity bits. Then, the following parity prediction relationships hold:

$$p(02 \times A \bmod G) = a_{n-1} p_G + p_A \tag{11}$$

$$p(03 \times A \bmod G) = a_{n-1} p_G, \tag{12}$$

where, for brevity, $A(x)$ and $G(x)$ are shortened as A and G, respectively.

Proof. First, we prove (11). Recall that $02 = x$ (see Section 2.3),

$$02 \times A \bmod G = \left(x \times \sum_{i=0}^{n-1} a_i x^i \right) \bmod G$$

$$= \left(\sum_{i=0}^{n-1} a_i x^{i+1} \right) \bmod G = \left(a_{n-1} x^n + \sum_{i=0}^{n-2} a_i x^{i+1} \right) \bmod G$$

and, since reducing this expression modulus G means setting $G = x^n + \sum_{i=0}^{n-1} g_i x^i = 0$, i.e., $x^n = \sum_{i=0}^{n-1} g_i x^i$, we obtain:

$$\left(a_{n-1} x^n + \sum_{i=0}^{n-2} a_i x^{i+1} \right) \bmod G = a_{n-1} \sum_{i=0}^{n-1} g_i x^i + \sum_{i=0}^{n-2} a_i x^{i+1}$$

$$= a_{n-1} \left(g_0 + \sum_{i=1}^{n-1} g_i x^i \right) + \sum_{i=0}^{n-2} a_i x^{i+1}$$

$$= a_{n-1} g_0 + a_{n-1} \sum_{i=1}^{n-1} g_i x^i + \sum_{i=0}^{n-2} a_i x^{i+1}$$

$$= a_{n-1} g_0 + \sum_{i=0}^{n-2} a_{n-1} g_{i+1} x^{i+1} + \sum_{i=0}^{n-2} a_i x^{i+1}$$

$$= a_{n-1} g_0 + \sum_{i=0}^{n-2} (a_{n-1} g_{i+1} + a_i) x^{i+1}.$$

We next calculate the parity of the product $02 \times A \bmod G$:

$$p(02 \times A \bmod G) = p\left(a_{n-1}g_0 + \sum_{i=0}^{n-2}(a_{n-1}g_{i+1} + a_i)x^{i+1}\right)$$
$$= a_{n-1}g_0 + \sum_{i=0}^{n-2}(a_{n-1}g_{i+1} + a_i)$$
$$= a_{n-1}g_0 + a_{n-1}\sum_{i=0}^{n-2} g_{i+1} + \sum_{i=0}^{n-2} a_i$$
$$= a_{n-1}g_0 + a_{n-1}\sum_{i=1}^{n-1} g_i + \sum_{i=0}^{n-2} a_i = a_{n-1}\left(g_0 + \sum_{i=1}^{n-1} g_i\right) + \sum_{i=0}^{n-2} a_i$$
$$= a_{n-1}\sum_{i=0}^{n-1} g_i + \sum_{i=0}^{n-2} a_i = a_{n-1}\left(\sum_{i=0}^{n-1} g_i + 1 + 1\right) + \sum_{i=0}^{n-2} a_i + a_{n-1} + a_{n-1} = a_{n-1}\left(\sum_{i=0}^{n-1} g_i + 1 + 1\right) + \sum_{i=0}^{n-1} a_i + a_{n-1}.$$

Substituting $p_G = \sum_{i=0}^{n-1} g_i + 1$ and $p_A = \sum_{i=0}^{n-1} a_i$ we obtain:

$$a_{n-1}\left(\sum_{i=0}^{n-1} g_i + 1 + 1\right) + \sum_{i=0}^{n-1} a_i + a_{n-1}$$
$$= a_{n-1}(p_G + 1) + p_A + a_{n-1}$$
$$= a_{n-1}p_G + a_{n-1} + p_A + a_{n-1} = a_{n-1}p_G + p_A.$$

This completes the proof of (11).

To prove (12), note that $03 = 02 + 01 \bmod G$ and, since the parity operator $p()$ is linear it follows:

$$p(03 \times A \bmod G) = p(02 \times A + A \bmod G)$$
$$= p(02 \times A \bmod G) + p(A) = a_{n-1}p_G + p_A + p_A = a_{n-1}p_G.$$

This completes the proof of (12). □

For AES and the finite field $GF(2^8)$, (11) and (12) can be further simplified. We set $n = 8$ and observe that the parity of the AES field generator polynomial $\Phi(x)$ (see (2)) is $p(\Phi(x)) = 1$. Substituting these two equalities in (11) and (12), we obtain:

$$p(02 \otimes A) = a_7 + p(A) \tag{13}$$

$$p(03 \otimes A) = a_7. \tag{14}$$

Incidentally, note that every generator polynomial irreducible over the coefficient field $GF(2)$ must have an odd parity; otherwise, it would have the factor $x + 1$. Therefore, the above holds for every binary finite field of type $GF(2^n)$, independent of the representation chosen for the field. Note also that (14) might be rewritten using the input parity bit p_A, as follows: $p(03 \otimes A) = \sum_{i=0}^{6} a_i + p(A)$. However, this form is more expensive than form (14) since it contains more modulus 2 additions (i.e., more XOR gates). The complete parity prediction scheme of MixColumns can now be described.

Lemma A.2 (Parity of MixColumns). *Consider the MixColumns transformation given by mappings (6)-(9). Denote by $p_{r,c}$ the parity bit of the byte element $s_{r,c}$ and by $s_{r,c}^{(i)}$ the ith bit of the byte element $s_{r,c}$; $0 \le r, c \le 3$. Then, the predicted parity bits of MixColumns are:*

$$p_{0,c} \mapsto p_{0,c} + p_{2,c} + p_{3,c} + s_{0,c}^{(7)} + s_{1,c}^{(7)} \tag{15}$$

$$p_{1,c} \mapsto p_{0,c} + p_{1,c} + p_{3,c} + s_{1,c}^{(7)} + s_{2,c}^{(7)} \tag{16}$$

$$p_{2,c} \mapsto p_{0,c} + p_{1,c} + p_{2,c} + s_{2,c}^{(7)} + s_{3,c}^{(7)} \tag{17}$$

$$p_{3,c} \mapsto p_{1,c} + p_{2,c} + p_{3,c} + s_{3,c}^{(7)} + s_{0,c}^{(7)}. \tag{18}$$

Sketch of Proof. We first prove mapping (15). Using mapping (6) and the fact that the parity operator $p()$ is linear, we obtain:

$$p_{0,c} \mapsto p(\alpha \otimes s_{0,c} \oplus \beta \otimes s_{1,c} \oplus s_{2,c} \oplus s_{3,c})$$
$$= p(02 \otimes s_{0,c}) + p(03 \otimes s_{1,c}) + p(s_{2,c}) + p(s_{3,c})$$
$$= s_{0,c}^{(7)} + p(s_{0,c}) + s_{1,c}^{(7)} + p(s_{2,c}) + p(s_{3,c})$$
$$= p_{0,c} + p_{2,c} + p_{3,c} + s_{0,c}^{(7)} + s_{1,c}^{(7)}.$$

The remaining mappings (16)-(18) can be proven similarly and the proofs are omitted for the sake of brevity. □

APPENDIX B

PARITY PREDICTION IN INVMIXCOLUMNS

The InvMixColumns transformation is readily obtained by inverting MixColumns. Since MixColumns is linear (see mappings (6)-(9)), it can be represented in a matrix form with elements over $GF(2^8)$. A similar representation exists for InvMixColumns by taking the inverse matrix in $GF(2^8)$. We observe that the InvMixColumns transformation uses more coefficients than MixColumns, namely: 09, $0b$, $0d$, and $0e$. Lemma A.1 can be extended to this set of four coefficients, yielding the following parity prediction equations:

$$p(09 \otimes A) = a_5 + a_6 + a_7 \tag{19}$$

$$p(0b \otimes A) = a_5 + a_6 + p(A) \tag{20}$$

$$p(0d \otimes A) = a_5 + p(A) \tag{21}$$

$$p(0e \otimes A) = a_5 + a_7 + p(A). \tag{22}$$

The proof of these equations is similar to the proof shown above for the coefficients 02 and 03. We may then rephrase Lemma A.2 for InvMixColumns, obtaining the necessary equations for predicting the parity bits of the outputs of InvMixColumns. The resulting equations have the same kind of structure as those for MixColumns and are only slightly more complicated. These equations are omitted for the sake of brevity.

As already observed before regarding (14), it is possible to eliminate from (19)-(22) the dependence on the input parity bit. For instance, (20) may be rewritten as: $p(0b \otimes A) = a_0 + a_1 + a_2 + a_3 + a_4 + a_7$. However, such a form has a hardware cost higher than that of (20). Equations (13)-(14) and (19)-(22) have minimum hardware cost.

APPENDIX C
SINGLE FAULT COVERAGE—PROOF

In this appendix, we show that the parity code scheme described in Section 5.2.2 achieves, under conditions to be stated, a 100 percent detection coverage with respect to single bit faults in the encryption module.

C.1 Fault and Error Dispersion Model

The following fault model is assumed: one single bit error injected in the encryption module during the encryption process, at the inputs of one of the four round transformations. It is also assumed that the key schedule module is fault-free, hence the key material is error-free. It is further assumed that the parity bits are checked at the end of the 10th round. Note that the above fault model does not consider the possibility of faults occurring in the internal circuitry of the submodules performing each transformation. Recall that each byte element $s_{r,c}$ of the state matrix S is associated with a parity bit $p_{r,c}$, for every $0 \leq r, c \leq 3$. Define an error bit $e_{r,c}$ as follows: $e_{r,c} = p(s_{r,c}) + p_{r,c}$, for $0 \leq r, c \leq 3$. Since even parity is used, the value $e_{r,c} = 0, 1$ indicates whether the 9-bit sequence $s_{r,c}, p_{r,c}$ has a correct (even) or incorrect (odd) parity status, respectively. In order to study the propagation of incorrect parity states during the rounds, we define a 4×4 error state matrix E, as follows:

$$E = \begin{bmatrix} e_{0,0} & e_{0,1} & e_{0,2} & e_{0,3} \\ e_{1,0} & e_{1,1} & e_{1,2} & e_{1,3} \\ e_{2,0} & e_{2,1} & e_{2,2} & e_{2,3} \\ e_{3,0} & e_{3,1} & e_{3,2} & e_{3,3} \end{bmatrix}.$$

To investigate the dispersion of errors in the encryption process, it suffices to examine how the round transformations modify the error state matrix E.

SubBytes (or Sbox). The error status of each input byte element propagates unaltered through Sbox. Note that the number and positions of the erroneous bits may change from the input to the output bytes, but their parity states remain the same. Hence, Sbox maps the error state matrix E to itself.

AddRoundKey. The behavior is the same as above since the round key is assumed to be error-free.

The remaining two round transformations behave in a more complex way.

ShiftRows. The parity states of the input bytes are preserved but rotated, according to mapping (5).

MixColumns. MixColumns is the most complex round transformation; it modifies the error state matrix E as follows:

$$e_{0,c} \mapsto e_{0,c} + e_{2,c} + e_{3,c} \tag{23}$$

$$e_{1,c} \mapsto e_{0,c} + e_{1,c} + e_{3,c} \tag{24}$$

$$e_{2,c} \mapsto e_{0,c} + e_{1,c} + e_{2,c} \tag{25}$$

$$e_{3,c} \mapsto e_{1,c} + e_{2,c} + e_{3,c} \tag{26}$$

for every $0 \leq c \leq 3$.

To explain the above mapping, notice that the topmost line is a direct consequence of mapping (15), which states that $p_{0,c} \mapsto p_{0,c} + p_{2,c} + p_{3,c} + s^{(7)}_{0,c} + s^{(7)}_{1,c}$. The remaining three mapping relations can be justified in a similar manner. In summary, only ShiftRows and MixColumns determine the propagation of parity errors during the rounds and both behave in a linear way.

C.2 Proof of the Coverage

We prove now that the proposed parity-based detection technique achieves a 100 percent coverage for single bit faults in the encryption module. The key issue is whether an incorrect parity status of a data byte may be masked at a later step of the encryption process. Assuming the same fault model as above, we observe that the dispersion of errors in the error state and, hence, error masking, depend only on the round transformations ShiftRows and MixColumns. Furthermore, since ShiftRows is a rotation of the error state, it cannot cause error masking as the number of entries of value 1 in E is not changed. In summary, error masking (if any) is due to MixColumns alone.

Therefore, a formal analysis of the way MixColumns modifies the error state E will reveal the error detection capabilities. From mappings (23)-(26), it is clear that MixColumns is a linear transformation and, hence, we can use matrices for its representation:

$$E_c \mapsto ME_c,$$

where

$$E_c = \begin{bmatrix} e_{0,c} \\ e_{1,c} \\ e_{2,c} \\ e_{3,c} \end{bmatrix} \quad \text{and} \quad M = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

for every column E_c ($0 \leq c \leq 3$) of the error state E. We next prove the following statement:

Statement C.1 (100 Percent Coverage). *The parity-based* EDC *detects single bit faults in the encryption module with 100 percent coverage.*

Proof. Single bit faults injected in the encryption module will not be detected only if error masking occurs. Based on the discussion above, error masking may occur only in MixColumns if the equality $ME_c = O$ holds, where O is the null error state column vector. It is well-known that a linear homogeneous system of equations $MX = O$, where X is a column vector of four variables over $GF(2)$, will admit a nonnull solution, i.e., $X \neq O$, only if the matrix M is singular, i.e., $\det(M) = 0$. But, a direct computation (here omitted) shows that $\det(M) = 1$. To

justify this, notice that M is an orthogonal matrix.[3] Hence, $ME_c \neq O$ for every nonnull error state column vector E_c and, therefore, MixColumns cannot mask errors. In conclusion, masking of single bit faults never occurs in the encryption module. □

APPENDIX D

HARDWARE OVERHEAD OF THE FAULT DETECTION PARITY CODES

In this appendix, we estimate the hardware overhead of the parity code-based fault detection scheme presented. Since many different implementations of AES are possible, some assumptions are made. Only the encryption module is considered since the decryption and key schedule algorithms have very similar parity prediction schemes; the size of the secret key is 128 bits; every round transformation is computed in parallel. The SubBytes round transformation is stored in a ROM. The other round transformations are implemented in a purely combinatorial way. Only XOR and 2-input AND logic gates are used for designing the round transformations (other than SubBytes) and the parity checkers. These assumptions yield the worst-case overhead. In fact, the prediction of the parity bits can be performed in a sequential manner, thus reusing some logic circuits, reducing the hardware overhead. The costs are estimated in terms of the number of XOR and 2-input AND logic gates. We must emphasize that, in practice, the overheads are largely dominated by those of SubBytes since this round transformation is computed by using 16 ROMs in parallel. The area of Sbox increases slightly more than 12.5 percent. The cost of ShiftRows is increased 12.5 percent, while MixColumns and AddRoundKey are increased 10.2 percent and 12.5 percent, respectively. Finally, the overhead of the parity checkers is much lower than 20 percent.

Therefore, a very conservative conclusion is that the total hardware overhead of the fault detection scheme for AES based on the parity code is approximately 10-20 percent.

ACKNOWLEDGMENTS

The work of Israel Koren has been supported in part by DARPA/AFRL NEST program under contract number F33615-02-C-4031. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the US Defense Advanced Projects Agency, AFRL, or the US Government.

REFERENCES

[1] G. Bertoni, L. Breveglieri, I. Koren, and V. Piuri, "Fault Detection in the Advanced Encryption Standard," *Proc. Conf. Massively Parallel Computing Systems (MPCS '02),* pp. 92-97, 2002.

[2] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, and V. Piuri, "On the Propagation of Faults and Their Detection in a Hardware Implementation of the Advanced Encryption Standard," *Proc. Int'l Conf. Application-Specific Systems, Architectures, and Processors (ASAP '02),* pp. 303-312, 2002.

[3] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, and V. Piuri, "A Parity Code Based Fault Detection for an Implementation of the Advanced Encryption Standard," *Proc. IEEE Int'l Symp. Defect and Fault Tolerance in VLSI Systems (DFT '02),* pp. 51-59, 2002.

[4] NIST, "Announcing the ADVANCED ENCRYPTION STANDARD (AES)," Federal Information Processing Standards Publication, no. 197, 26 Nov. 2001.

[5] B. Gladman, "A Specification for Rijndael, the AES Algorithm," http://fp.gladman.plus.com/, 2001.

[6] M. Akkar and C. Giraud, "Implementation of DES and AES, Secure against Some Attacks," *Proc. Workshop Cryptographic Hardware and Embedded Systems (CHES '01),* pp. 315-325, 2001.

[7] M. McLoone and J. McCanny, "High Performance Single-Chip FPGA Rijndael Algorithm Implementations," *Proc. Workshop Cryptographic Hardware and Embedded Systems (CHES '01),* pp. 68-80, 2001.

[8] V. Fischer and M. Drutarovsky, "Two Methods of Rijndael Implementation in Reconfigurable Hardware," *Proc. Workshop Cryptographic Hardware and Embedded Systems (CHES '01),* pp. 81-96, 2001.

[9] H. Kuo and I. Verbauwhede, "Architectural Optimization for a 1.82Gbits/sec VLSI Implementation of the AES Rijndael Algorithm," *Proc. Workshop Cryptographic Hardware and Embedded Systems (CHES '01),* pp. 53-67, 2001.

[10] A. Rudra, P. Dubey, C. Jutla, V. Kumar, J. Rao, and P. Rohatgi, "Efficient Rijndael Encryption Implementation with Composite Field Arithmetic," *Proc. Workshop Cryptographic Hardware and Embedded Systems (CHES '01),* pp. 175-188, 2001.

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

[12] J. Daemen and V. Rijmen, "The Block Cipher Rijndael," *Smart-Card Research and Applications,* J. Quisquater and B. Schneier, eds., pp. 288-296, Springer-Verlag, 2000.

[13] D. Whiting, B. Schneier, and S. Bellovin, "AES Key Agility Issues in High-Speed IPsec Implementations," Counterpane Internet Security, http://www.counterpane.com/aes-agility.html, 2000.

[14] E. Biham, "A Fast New DES Implementation in Software," *Proc. Int'l Symp. Foundations of Software Eng. (FSE '97),* pp. 260-273, 1997.

[15] H. Eberle, "A High-Speed DES Implementation for Network Application," *Proc. Int'l Conf. Cryptology (CRYPTO '92),* pp. 521-539, 1993.

[16] R. Lidl and H. Niederreiter, *Introduction to Finite Fields and Their Applications.* Cambridge Univ. Press, 1986.

[17] Nat'l Bureau of Standards, "Data Encryption Standard,"US Dept. of Commerce, FIPS pub. no. 46, Jan. 1977.

Guido Bertoni received the DrEng degree in computer engineering from the Politecnico di Milano in 1999. Since 2000, he has been a PhD student at the Politecnico di Milano. The topic of the PhD program is the efficient implementation of cryptographic algorithms in embedded systems.

Luca Breveglieri received the DrEng degree in electronic engineering and the PhD degree in electronic engineering of information technology and systems from the Politecnico di Milano, Italy, in 1986 and 1992, respectively. From 1991 to 1998, he was a computer technician and a part-time researcher in the Dipartimento di Elettronica e Informazione (DEI), Politecnico di Milano (PdM). Since 1998, he has been an associate professor of computer science at the Politecnico di Milano. While a computer technician, he worked as a computer systems and networks manager, administering the computers and the local area network of DEI. His research interests include architectures of computing systems and networks, application specific VLSI synthesis, computer arithmetic and cryptography, and formal languages theory.

3. The sum of the squares of the bits of each row of M is always 1 and the sum of the bit-wise products of each pair of different rows of M is always 0; of course, all the operations must be computed modulus 2.

Israel Koren (S'72-M'76-SM'87-F'91) received the BSc, MSc, and DSc degrees from the Technion-Israel Institute of Technology, Haifa, in 1967, 1970, and 1975, respectively, all in electrical engineering. He is currently a professor of electrical and computer engineering at the University of Massachusetts, Amherst. Previously he was with the Technion-Israel Institute of Technology. He also held visiting positions with the University of California at Berkeley, University of Southern California, Los Angeles, and University of California, Santa Barbara. He has been a consultant to several companies, including IBM, Intel, Analog Devices, AMD, Digital Equipment Corp., National Semiconductor, and Tolerant Systems. Dr. Koren's current research interests include techniques for yield and reliability enhancement, fault-tolerant architectures, real-time systems, and computer arithmetic. He has published extensively in several IEEE Transactions and has more than 170 publications in refereed journals and conferences. He currently serves on the editorial board of the *IEEE Transactions on VLSI Systems*. He was a co-guest editor for the *IEEE Transactions on Computers*, special issue on high yield VLSI systems, April 1989, and the special issue on computer arithmetic, July 2000, and served on the editorial board of these transactions between 1992 and 1997. He also served as general chair, program chair, and program committee member for numerous conferences. He has edited and coauthored the book *Defect and Fault-Tolerance in VLSI Systems*, volume 1 (Plenum, 1989). He is the author of the textbook *Computer Arithmetic Algorithms* (A.K. Peters, Ltd., 2002). He is a fellow of the IEEE.

Paolo Maistri received the MS degree in electrical engineering and computer science from the University of Illinois at Chicago in 2001; he received the DrEng degree in computer engineering from the Politecnico di Milano later in the same year. He is currently an assistant researcher at the Politecnico di Milano. His research interests include fault tolerance, computer arithmetic, and cryptography.

Vincenzo Piuri received the PhD degree in computer engineering in 1989 from the Politecnico di Milano, Italy. From 1992 to September 2000, he was an associate professor of operating systems at the Politecnico di Milano. Since October 2000, he has been a full professor in computer engineering at the University of Milano, Italy. He was a visiting professor at the University of Texas at Austin during the summers from 1993 to 1999. His research interests include distributed and parallel computing systems, computer arithmetic, application-specific processing architectures, digital signal processing architectures, fault tolerance, neural network architectures, theory and industrial applications of neural techniques for identification, prediction, control, signal and image processing. Original results have been published in more than 150 papers in book chapters, international journals, and proceedings of international conferences. He is a fellow of the IEEE and a member of the ACM, INNS, AEI. He is an associate editor of the *IEEE Transactions on Neural Networks*.