A Parity Code Based Fault Detection for an Implementation of the Advanced Encryption Standard

Guido Bertoni[1], Luca Breveglieri[1], Israel Koren[2], Paolo Maistri[1], Vincenzo Piuri[3]

[1]Department of Electronics and Information Technology
Politecnico di Milano, Milano, ITALY
[2]Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA, USA
[3]Department of Information Technologies, University of Milan, Crema, ITALY

bertoni@elet.polimi.it, brevegli@elet.polimi.it,
koren@ecs.umass.edu, maistri@elet.polimi.it, piuri@dti.unimi.it

Abstract

Concurrent fault detection for a hardware implementation of the Advanced Encryption Standard (AES) *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 present a novel fault detection scheme which is based on a multiple parity bit code and show that the proposed scheme leads to very efficient and high coverage fault detection. We then estimate the associated hardware costs and detection latencies.*

1: Introduction

The Rijndael Advanced Encryption Standard (AES) algorithm is a secret-key crypto-system recently approved as standard by NIST [4]. AES is intended to replace the widely used DES and Triple-DES crypto-systems, due to their limited level of security [3, 5]. AES is considerably more complex than the DES crypto-system it replaces and consequently, any implementation of AES is expected to be subject to a larger number of fault occurrences compared to DES. Recent work has shown that even a single transient fault occurring during the AES encryption (or decryption) process will very likely result in a large number of errors in the encrypted/decrypted data [1, 2]. Such faults must be detected before transmission to avoid the transmission and use of erroneous encrypted/decrypted data. Moreover, fault detection and possibly fault tolerance is a desirable property for preventing malicious attacks, aimed at extracting sensitive information from the device, like the secret key [5, 9].

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 [9] from the perspective of preventing attacks based on malicious injection of faults into the AES crypto-processor. Karri's 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 attack-resistant architecture. These solutions have varied detection time latencies and hardware costs. In general, such solutions exhibit a large cost, being close to duplication.

In this paper, a novel fault detection technique for AES is presented and evaluated. The proposed technique is based on an error detection code and as such has a lower overhead. Our results show that it is practical to devise efficient fault detection codes even for a complex and non-homogeneous algorithm like AES.

The paper is organized as follows. In Section 2 a brief overview of the AES algorithm is presented, and the implementation details which are necessary for understanding our proposed error detection scheme are included. In Section 3 our fault detection algorithm is described. Details regarding the proposed algorithm, the expected fault coverage and its hardware overhead are omitted for brevity, or they are briefly presented when the context needs to be clarified. We present our conclusions in Section 4.

2: The Rijndael algorithm

The Rijndael AES is a secret-key (symmetric) block cipher crypto-system [4]. The encryption algorithm accepts one data block (or plaintext) and the key, and produces the encrypted data block (the input and output data blocks are of identical size). The decryption algorithm accepts one encrypted data block and the key, and outputs the plaintext. 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 regarded 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 somewhat flexible crypto-system. In principle, the allowed sizes of the data block and the secret key are all the combinations of 128, 192 and 256 bits. However, NIST has restricted the size of the data blocks to 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 key size. For the three key sizes of 128, 192 and 256 bits, a number of 10, 12 and 14 rounds is required, respectively, plus an initial special round (called round 0). The block diagram of the Encryption process is shown in Figure 1 in [1].

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 (also called Sbox), ShiftRows, MixColumns and AddRoundKey, see again Figure 1 from [1]. The transformation AddRoundKey is the point where the secret key enters the 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.

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 the 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 which, for a key size of 128 bits, is basically iterative, though not completely regular. The reader is referred to [4] 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* [10]. 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.

3: Fault detection technique

A proposal for error detection in the data-path of AES was described in [9]. The proposed technique was based on the use of redundancy and was also discussed in [1, 2]. In this section we present a novel fault detection technique which is based on error detecting codes, namely a suitably designed multiple parity bit code. This technique proves to be very efficient and has a rather low hardware overhead.

Our objective is to develop a fault detection technique which will be independent of the particular hardware implementation chosen. In what follows we assume that 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); each module executes the corresponding algorithm (see Section 2). All these 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.

3.1: 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 non-linear algorithm. 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.

3.1.1: 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 can not, however, employ only a single parity bit for fault detection in the AES for the following reasons:

- As shown in [1, 2], errors spread quickly throughout the data block as encryption goes on, and on the average about one half of the state bits become corrupt. Hence, the fault coverage of the parity code would be at best around 50%, 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 has a global dependence on all information bits.

To circumvent these problems, we propose to associate one parity bit with each byte element of the state matrix S (see equation 1.4.1 in [4]), for a total of 16 parity bits. These parity bits can be ideally 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 is even. The state and parity matrices are shown below where $p_{r,c}$ is the parity bit of the state byte $s_{r,c}$ (for $r, c = 0, 1, 2, 3$).

$$\begin{bmatrix} p_{0,0} & p_{0,1} & p_{0,2} & p_{0,3} \\ p_{1,0} & p_{1,1} & p_{1,2} & p_{1,3} \\ p_{2,0} & p_{2,1} & p_{2,2} & p_{2,3} \\ p_{3,0} & p_{3,1} & p_{3,2} & p_{3,3} \end{bmatrix} \qquad \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} \tag{1}$$

The parity matrix certainly allows 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 leads to a considerable reduction in the complexity of the parity prediction process.

To implement this coding scheme it is necessary:

- For each round transformation, to develop a method for predicting the output parity, given the input state and the input parity.
- To schedule checkpoints during the encryption process. At least one checkpoint is required, but possibly two or more can be included 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.

3.1.2: 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 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 to replace 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 errors 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 can be expected, we may 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 may increase the detection capabilities of this scheme by adding one (or more) correct output data bits 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, i.e.,

$$\begin{bmatrix} p_{0,0} & p_{0,1} & p_{0,2} & p_{0,3} \\ p_{1,0} & p_{1,1} & p_{1,2} & p_{1,3} \\ p_{2,0} & p_{2,1} & p_{2,2} & p_{2,3} \\ p_{3,0} & p_{3,1} & p_{3,2} & p_{3,3} \end{bmatrix} \mapsto \begin{bmatrix} p_{0,0} & p_{0,1} & p_{0,2} & p_{0,3} \\ p_{1,1} & p_{1,2} & p_{1,3} & p_{1,0} \\ p_{2,2} & p_{2,3} & p_{2,0} & p_{2,1} \\ p_{3,3} & p_{3,0} & p_{3,1} & p_{3,2} \end{bmatrix} \tag{2}$$

MixColumns: The prediction of the output parity bits of MixColumns is the most mathematically complex one. The detailed solution is omitted for brevity. The final set of equations for predicting the parity bits are, however, quite simple: see equations (3), (4) and (5), where $s^{(7)}_{r,c}$ is the most significant bit of the byte at row r and column c:

$$02 \times p_{r,c} \mapsto p_{r,c} + s^{(7)}_{r,c} \tag{3}$$

$$03 \times p_{r,c} \mapsto s^{(7)}_{r,c} \tag{4}$$

$$\begin{bmatrix} p_{0,c} \\ p_{1,c} \\ p_{2,c} \\ p_{3,c} \end{bmatrix} \mapsto \begin{bmatrix} p_{0,c} + p_{2,c} + p_{3,c} + s^{(7)}_{0,c} + s^{(7)}_{1,c} \\ p_{0,c} + p_{1,c} + p_{3,c} + s^{(7)}_{1,c} + s^{(7)}_{2,c} \\ p_{0,c} + p_{1,c} + p_{2,c} + s^{(7)}_{2,c} + s^{(7)}_{3,c} \\ p_{1,c} + p_{2,c} + p_{3,c} + s^{(7)}_{3,c} + s^{(7)}_{0,c} \end{bmatrix} \tag{5}$$

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. For each element of the status matrix:

$$p_{r,c} \mapsto p_{r,c} + p^{(k)}_{r,c} \tag{6}$$

where $p^{(k)}$ is the parity bit associated to the key material.

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.

It is also necessary to decide on the scheduling of the parity checks. 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 somewhat lower fault coverage.

In Section 3.1.4 we show that both policies (2) and (3) reach a sufficiently high fault coverage, namely very close to 100%, for the single faulty bit model. This is a non-obvious result, since the behavior of AES, as described in [1, 2], 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; the prediction scheme for the inverse round transformation InvMixColumns, which is the most complex of the four, can be obtained by applying the same approach used for the MixColumn operation. The same approach (with few adjustments) also works for the (Inverse) Key Schedule module, as this module employs the same basic operations as the Encryption (and Decryption) module.

3.1.3: Cost of the proposed parity code

The cost in terms of hardware overhead for the parity-based EDC described above is limited. In Table1 it is shown that this overhead falls in the range 10% – 20% with respect to the nominal hardware cost of the Encryption module, for the two checking policies (2) and (3) described in Section 3.1.2. 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 constraint of the application.

Module	**hardware Overhead**
SubBytes (Sbox)	slightly more than 12.5%
ShiftRows	12.5%
MixColumns	10.2%
AddRoundKey	12.5%
Error detection	less than 20.5%

Table 1. *Hardware overheads of the parity-based fault detection scheme for* AES

These fault coverage and cost figures are likely to extend - with only 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 3.1, 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.

3.1.4: 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 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 have been performed with data block and key of 128 bits.

1. 5000 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 $5000 \times 10 \times 128 = 6.4 \times 10^6$. All these tests used the same secret key. Our parity bits scheme detected all the faults.
2. 5000 secret keys were randomly selected and used with the same 128-bit data block. The same single bit errors as in (1) were injected for a total of 6.4×10^6 tests. Here too, all the faults were detected by our parity scheme.
3. 100 random secret keys and 1000 random data block were selected and every data block was encrypted with each secret key. 1280 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 tenth round. In the next type of tests the parity check was instead done at the end of the round.

4. 5×10^5 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% fault coverage. The total number of tests of this type was $5 \times 10^5 \times 128 = 6.4 \times 10^7$.

The last two types of tests considered 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 was performed at the end of the (simplified) round. The observed fault coverage has again been 100%.

5. 256 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 $256 \times 8 = 2048$. All the faults were detected.
6. 1000 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 $1000 \times 128 = 1.28 \times 10^4$. Again, all the injected faults were detected.

These six types of tests strongly suggest that the parity-based EDC achieves a 100% 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 had to be omitted for the sake of brevity. In this proof, however, we only considered single faults injected at the beginning of an encryption round, at the inputs of one of the four round transformations.

Therefore, 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 and 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 between 2 and 16. Figure 1 shows the percentages of undetected faults, for 3 to 16 injected faults.

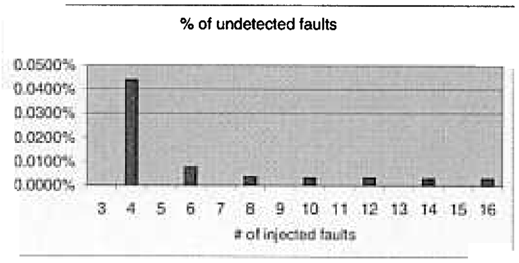

Figure 1. *Percentage of the undetected multiple faults injected in the Encryption module.*

For double faults, the observed percentage of undetected faults was 0.875% but is not shown in Figure 1 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%, 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 shown 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[1], clearly decreases with the order of the fault. This explains why the percentages in Figure 1 are decreasing. Due to the high dispersion of errors in AES (see [1, 2]), it is reasonable to expect that such a behavior remains essentially unchanged when the faults are injected between two round transformations.

In summary, the coverage of the parity-based EDC is very high and for single bit faults it reaches 100%. Asymptotically, the coverage converges to 99.997%. The apparent coverage of almost 100% for all multiple faults of odd order over 1 is worthy of further investigation, but this has been omitted for the sake of brevity.

4: Conclusions

A fault detection technique for a hardware implementation of the AES algorithm has been presented in this paper. The proposed technique, which is based on the use of parity codes, exhibits a very good fault coverage, limited hardware overhead cost and a short detection latency. 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, covering the various modules of AES. Most experimental results were also formally verified, but the proof had to be omitted for brevity.

[1]Or also into two or more data bytes, but distributed in sub-groups of even order each.

Acknowledgment: The work of Israel Koren has been supported in part by DARPA's NEST program.

References

[1] G. Bertoni, L. Breveglieri, I. Koren, V. Piuri, "Fault Detection in the Advanced Encryption Standard," *Proceedings of MPCS '02*, Ischia, Italy, 2002.

[2] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, V. Piuri, "On the Propagation of Faults and their Detection in a Hardware Implementation of the Advanced Encryption Standard," *Proceedings of ASAP '02*, San Jose, CA, USA, pp. 303-312, 2002.

[3] NIST, "Announcing the ADVANCED ENCRYPTION STANDARD (AES)," Federal Information Processing Standards Publication, n. 197, November 26, 2001.

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

[5] M. Akkar, C. Giraud, "An Implementation of DES and AES, Secure against some Attacks," *Proceedings of CHES '01*, pp. 315-325, 2001.

[6] M. McLoone, J. McCanny, "High Performance single-Chip FPGA Rijndael Algorithm Implementations," *Proceedings of CHES '01*, pp. 68-80, 2001.

[7] V. Fischer, M. Drutarovsky, "Two Methods of Rijndael Implementation in Reconfigurable Hardware," *Proceedings of CHES '01*, pp. 81-96, 2001.

[8] H. Kuo, I. Verbauwhede, "Architectural Optimization for a 1.82Gbits/sec VLSI Implementation of the AES Rijndael Algorithm," *Proceedings of CHES '01*, pp. 53-67, 2001.

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

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

[11] R. Lidl, H. Niederreiter, *Introduction to Finite Fields and their Applications*, Cambridge University Press, 1986.