On the Propagation of Faults and Their Detection in a Hardware Implementation of the Advanced Encryption Standard

Guido Bertoni[1] , Luca Breveglieri[1] , Israel Koren[2] ,
Paolo Maistri[1] and 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 Technology, 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

High reliability is a desirable property of any implementation of the Advanced Encryption Standard. To achieve high reliability all possible faults must be detected to avoid the use and transmission of erroneous encrypted/decrypted data. In this paper we first study the behavior of faults which may occur during the encryption and decryption procedures of AES, and the way such faults eventually propagate to the final result. We then describe an appropriate detection technique for these faults.

This work extends our preliminary results [1] *by considering more general fault models (e.g., permanent and multiple transient faults), and the possibility of fault masking.*

1. Introduction

The Rijndael AES algorithm (AES) is on its way to become the most widely used secret-key algorithm [3]. AES is intended to replace DES and Triple-DES due to their low level of security, see for instance [2] and [5]. It enhances the features of DES, extending them to work with longer keys and wider blocks of data. AES is a cryptographic system rooted in the algebra of Galois (or finite) fields, in particular fields of type $GF(2^n)$. Some implementations of AES, both in software and in hardware, have been recently proposed in [5], [6], [7], [8] and [9], but a well-defined and widespread solution does not yet exist, and consequently this is still a current topic for research. Some coprocessor VLSI architectures for computing AES have been proposed and evaluated at the simulation level (see the above references).

Fault tolerance is an important issue when designing a coprocessor architecture for implementing the AES algorithm. AES is in fact considerably more complex than the DES algorithm it is meant to replace. In a previous work [1] a preliminary study of fault detection and tolerance in AES has been carried out. Single transient faults were considered, and the diffusion of errors caused by a single transient fault was studied by software simulation. Some architectural suggestions were also discussed. The solution presented in [1] extends a partial solution presented in [10], which covers the data-path of the AES algorithm, but ignores the control part of AES, which is as important as the data-path part.

Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP'02)
1063-6862/02 $17.00 © 2002 **IEEE**

IEEE COMPUTER SOCIETY

The present paper completes the analysis and extends the proposals presented in [1]. More complex fault models like permanent and multiple transient faults are taken into consideration and the phenomenon of error masking in the presence of multiple faults is studied. Moreover, faults in the AES algorithm are modelled at a finer-grained level than was done in [1]. The architectural suggestions previously presented in [1] are briefly summarized and extended taking into consideration the new results.

The paper is organised as follows. In Section 2 a brief explanation of the Rijndael AES algorithm is presented. In Section 3 the results previously reported in [1] are briefly summarized. Sections 4 and 5 present and discuss the enhanced fault models for AES and their behaviour: inner faults, injected at a fine-grained level into AES; permanent and multiple faults, and error masking (or aliasing). The propagation of a fault is obtained by software simulations of the AES algorithm, in the presence of randomly injected bit errors. Section 6 reviews techniques for fault detection and provides some suggestions for their possible improvement.

2. Rijndael's AES algorithm

Rijndael is the algorithm adopted for AES. It is a symmetric block cipher algorithm, meaning that the same key is used for both encryption and decryption, and that it operates on blocks of data [3].

The AES Rijndael algorithm permits all the combinations of key sizes and data block sizes of 128, 196 and 256 bits. However, the National Institute of Standards and Technology (NIST) has restricted the length of the data blocks for the AES algorithm to 128 bits only, while the key size can be chosen as 128, 196 or 256 bits. We will consider the case where both the data block and the key are of length 128 bits.

AES has an iterative structure and works in rounds: a fixed set of operations, called a round, is iteratively applied to the input block of data, called state matrix (for our case it is a 4×4 matrix), the elements of which are bytes. Each round uses also a different round key. The round keys are progressively derived from the original key, a process called Key Scheduling. After iterating a predefined number of rounds the state matrix is output. The number of rounds to be executed is determined by each key size. A 128 bit key size requires 10 rounds with more rounds required for longer keys (see [3]).

The input block of data is encrypted by applying the required number of rounds. The inner rounds are identical and consist of four transformations each. The first and last rounds are shorter then the inner rounds. A brief overview of the algorithm, depicted in Figure 1, is given below, but the reader should refer to [3] for a complete description.

The four transformations forming each round are called SubByte, ShiftRow, MixColumn and KeyAddition. The SubByte transformation receives as input each element (one byte) of the state matrix and calculates its multiplicative inverse in the finite field $GF(2^8)$ (the AES standard specifies which reducing polynomial to use).

The ShiftRows transformation applies a different left rotation to each of the four rows of the state matrix. The MixColumns transformation treats each column as a 4-term polynomial, the coefficients of which are defined over the field $GF(2^8)$ (the reduction polynomial is the same as for SubByte), and multiplies each column by a constant polynomial, defined over the field $GF(2^8)$. The KeyAddition transformation computes the bit-wise XOR of the current round key and the state matrix.

To decrypt an enciphered block, the inverse transformation should be applied, and clearly, the secret key must be known in advance.

IEEE COMPUTER SOCIETY

Each round key is derived from the secret key by means of a special algorithm, called Key Scheduling. Eleven different round keys of 128 bits each are derived from a single secret key of 128 bits, for a total of 1408 bits of so-called "key material". Key Scheduling is based on combinations of byte permutations, additions and the application of SubByte. There are two methods for calculating the round keys: complete key unrolling and the so-called "on-the-fly" approach.

Key Scheduling is an invertible algorithm. From the current round key it is possible to reconstruct the previous round key. Invertibility is necessary for an efficient "on-the-fly" key scheduling during decryption, and will also be exploited for achieving error detection.

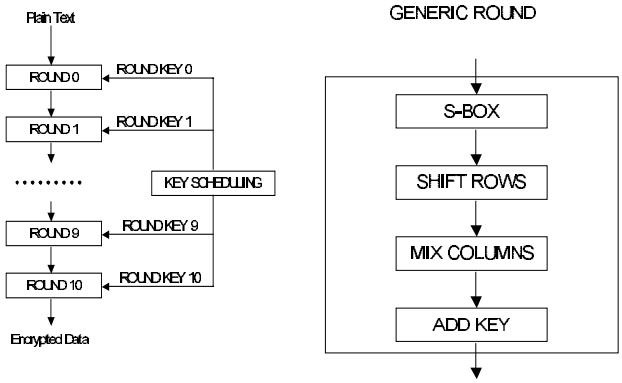

Figure 1: Logic scheme of AES with a 128- bit key.

3. Error propagation for single faults

In this section we briefly summarize the error propagation behavior of AES using a single faulty bit model, as was studied in [1]. The AES algorithm is basically divided into two parts: Key Scheduling, which uses the secret key to obtain the round keys, and Data-Path, which manipulates the plain text employing the round key to obtain the encrypted message (and vice versa for decryption). Key Scheduling can be considered as the control part of the AES algorithm. The effect of a single faulty bit may propagate and result in a large number of erroneous bits at the output. We have used a key size of 128 bits, but have verified that the observed behavior is similar for the other two admissible key sizes.

Figure 2 shows the results of our simulation experiments in which a faulty bit has been injected in the AES data-path. The fault is injected only at the beginning of the round, not during the inner transformations. The two graphs summarize the effect of a fault during the computation of the encryption and the effect on the result of the decryption, that is the number of erroneous bits caused by a single faulty bit injected at some round into the computation.

From Figure 2 we can see that a single faulty bit injected in the first round of encryption causes a high 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 AES is invertible. Moreover, corrupting a single bit in the intermediate result in any round between 1 and 8 yields a corrupt encrypted message, which is considerably different from the correct one. On the average, 64 output bits were erroneous. Note 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.

IEEE COMPUTER SOCIETY

Another part of the algorithm that can be affected by faults is the key scheduling. Two methods for key scheduling are possible: a complete key unrolling and an "on-the-fly" approach. The former can be subject to two types of errors: either a single faulty bit corrupting the stored unrolled key, or a faulty bit injected during the unrolling process which may spread over many bits. The "on-the-fly" approach can be subject only to the second type of error, because the unrolled key 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.

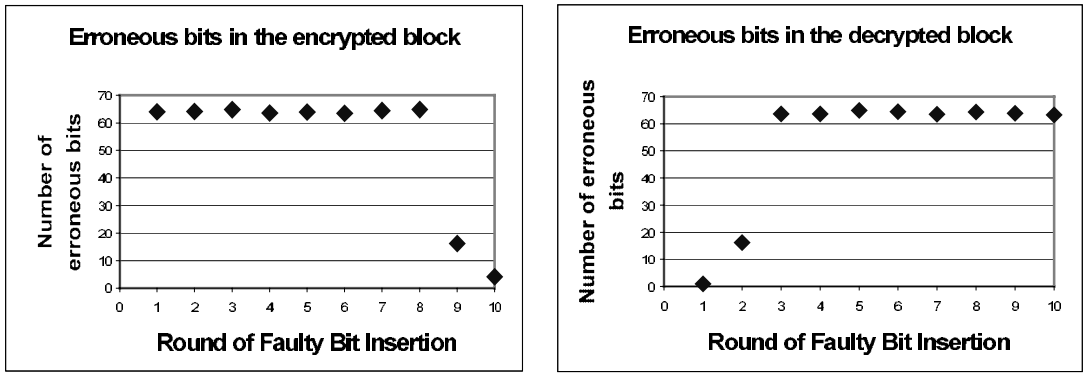

Figure 2: The mean number of erroneous bits in an encrypted and a decrypted data block.

Both situations have been simulated: a faulty bit in the stored unrolled key and a faulty bit injected during the process of unrolling the key. Note that a faulty bit in the stored unrolled key may also occur in a software-based implementation of AES (i.e., a bit flip in the memory location holding the unrolled key). Thus, part of our study applies to such implementations as well.

The complete key unrolling case is equivalent to the injection of a single error in the data-path. Corrupting a single bit in the stored unrolled key has an effect identical to injecting a faulty bit in the data block at the beginning of a round, since the round key is added to the state matrix at the end of each round.

As for the latter case, i.e., error injection and propagation during the key unrolling, we show in Figure 3 the number of erroneous bits obtained in the unrolled key, as a function of the number of the round where the fault has been injected. In this figure the average number of erroneous bits generated in the unrolled key for a secret key of 128 bits is shown, out of the total of 1408 bits composing the unrolled key material. Injecting a faulty bit into round 0 means receiving a corrupt secret key, hence this case is not considered. A faulty bit in the last unrolling round causes a single erroneous bit in the unrolled key.

The effect of a faulty bit in the key scheduling on the data-path is shown in Figure 4. The data-path is assumed to be fault-free, while the key scheduling is affected by the injection of a single faulty bit at some unrolling round. In this case it is possible to see that a faulty bit injected into the early rounds causes a high number of erroneous bits in the decryption process. If decryption is performed using the erroneous unrolled key to check for the presence of errors (as discussed in Section 4), it is not possible to detect the presence of a faulty bit in the key material. What happens is that the receiver will decrypt useless data and the sender will not be able to realize that the sent data was corrupted. This justifies the need for special attention to the fault management (i.e., fault detection and tolerance) of key unrolling.

IEEE COMPUTER SOCIETY

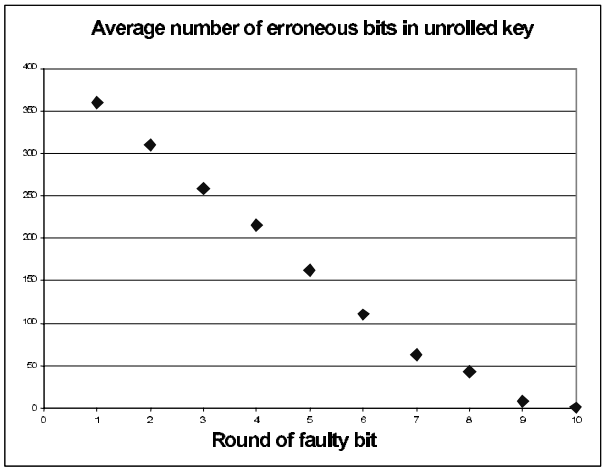

Figure 3: Propagation of a single faulty bit in the unrolling key process.

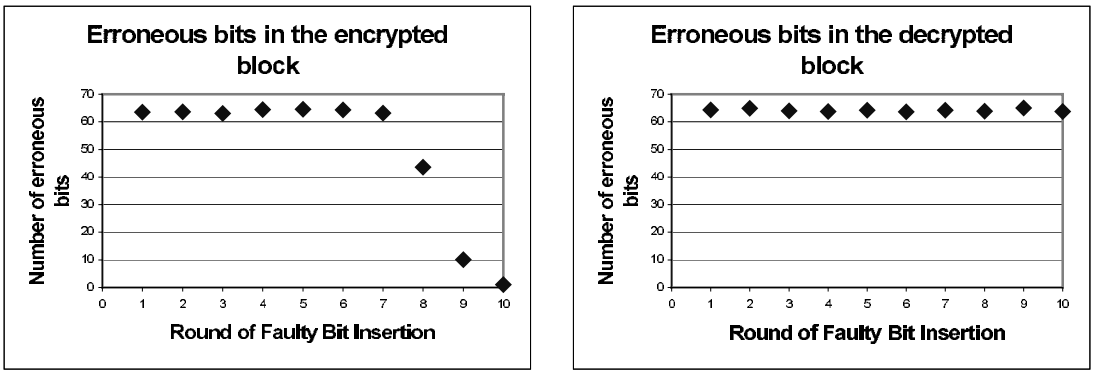

Figure 4: Effect of an erroneous unrolled key (due to a single faulty bit during the unrolling process) on the encryption and decryption processes.

4. Injection of internal faults

Injecting faults at the beginning of a round is simple to simulate, but does not cover all real life situations. Therefore, it is necessary to inject faults also during the internal transformations of a round. We start with the analysis of a single fault injection. We can easily deal with a fault injected during the very first round since this 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 Figure 5, 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 much more complicated and we must follow the errors as they propagate along the execution path.

The generic encryption round of AES consists of four operations: SubBytes, ShiftRows, MixColumns, AddKey. Their inverse operations are used in the decryption round. Hence, there are five spots where an error may be injected, from the beginning (before any other operation) to the end of the round. However, some conclusions may be inferred from the sequence of steps performed regarding the impact of a fault according to its position. A fault injected at the end of a generic round, provided that it is not the last round, is equivalent to an injection at the beginning of the following round, since no operations are performed in the middle. Thus, one position (spot) out of the possible five need not be analyzed.

IEEE COMPUTER SOCIETY

The propagation of a single fault injected in the other four positions will be influenced by the execution of the round components. The result may 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 AddKey components, where the error is only moved within the word or simply left untouched, respectively. The relative position of the fault with respect to these components does not affect the amount of errors in the output, since only linear operations like byte rotation and exclusive OR (XOR) are involved.

The two remaining components are non-linear, therefore they will greatly influence the propagation of errors. Figure 5 shows how the amount of erroneous bits changes as the position (spot) of fault injection is changed from round to round and within each round using a specific input and faulting each 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 spots most sensitive to faults precede the execution of SubBytes and MixColumn.

An analysis of the way these operations spread a single fault leads to some unexpected observations. The application of the Sbox substitution (SubBytes) creates a number of errors ranging from 1 to 8 and the most common value is 4 (Figure 6). Such data suggest that the number of erroneous output bits follow a binomial distribution, implying that the result would actually be random. A very small (non significant) value is obtained applying the Chi-square test to the frequencies given by the simulation, meaning 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.

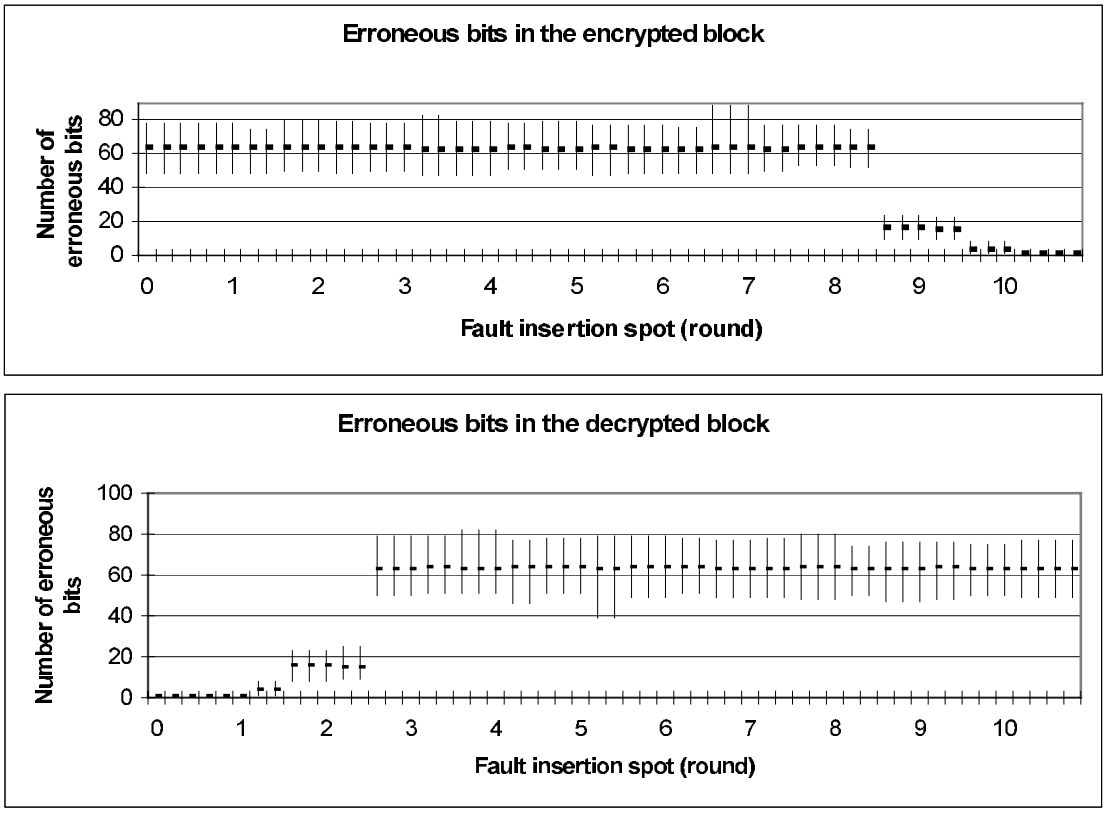

Figure 5: Effects of a transient fault in the state during encryption.

IEEE COMPUTER SOCIETY

The other non-linear component is the MixColumn operation: the distribution of the number of errors generated from a single incoming fault by this component is completely discrete. Injecting a fault before a MixColumn operation, we obtain either 5 or 11 errors at the output; similarly, injecting a fault in the InvMixColumn, we obtain a number of errors belonging to 11, 19, 21 or 23 (Figure 7). This behavior is due to the finite field multiplication performed by the routines. In fact, the MixColumn shows the same fault distribution pattern as the finite field product, although scaled in the number of bits involved (Figure 8). When the fault is in the most significant bit, it is spread by the reducing polynomial over a large number of bits, while the error is only shifted for every other bit. A similar pattern can be identified in the fault distribution of the single bit (Figure 9).

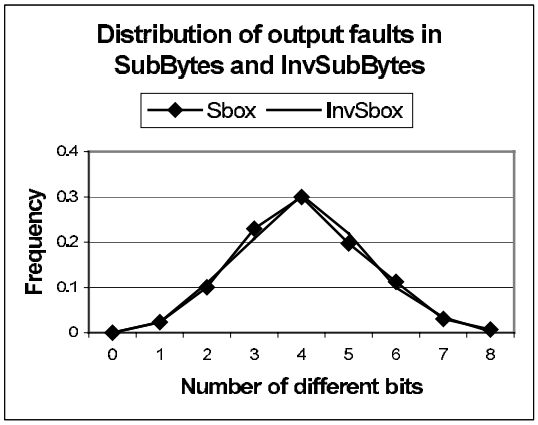

Figure 6: Effect of a single fault on the Sbox and InvSbox output.

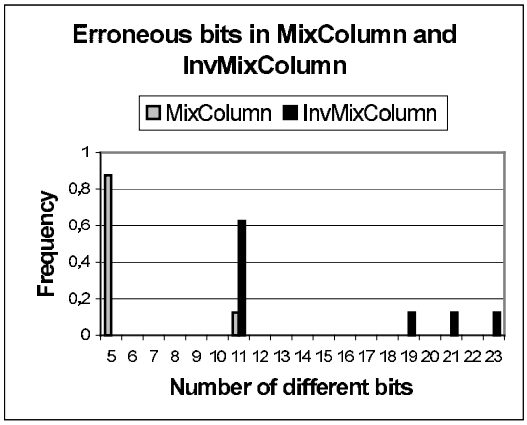

Figure 7: Effect of a single fault on the MixColumn and InvMixColumn operations.

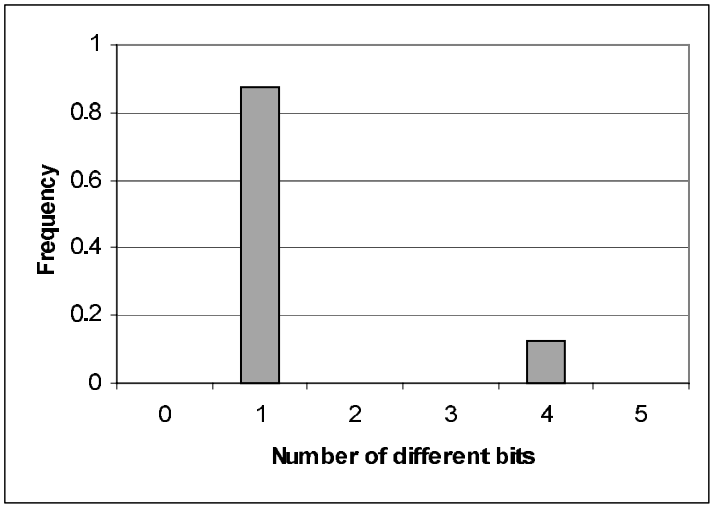

Figure 8: Effect of a fault injection on the finite field multiplication.

IEEE COMPUTER SOCIETY

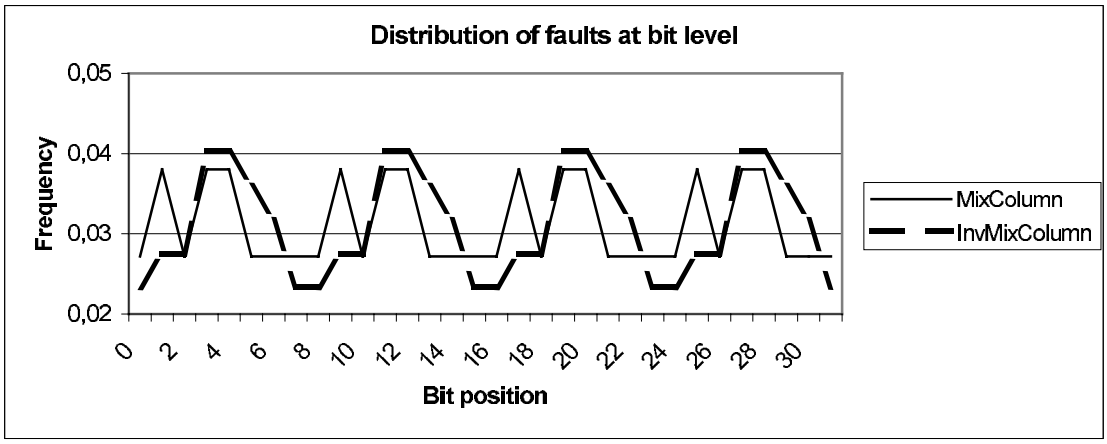

Figure 9: Distribution of the output error within the word in MixColumn and InvMixColumn operations.

5. Injection of multiple faults

In Section 4 we have considered the case when a single fault occurs in the computation affecting only one bit. In this section, instead, we want to 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 the encryption led to completely different outputs, both encrypted and decrypted (Figure 2). 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 times. Injecting two independent faults at different rounds shows similar results (Figure 10). 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 less erroneous bits) while for most cases the final output is random.

A permanent fault fixes the value of a specific bit to a constant 0 or 1, simulating a short or open circuit. This results in a variable number of injected faults, depending on the original bit value: in the worst case, it may add up to one error per 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 unrolled key table.

Our experiments with injecting multiple faults lead to two important observations. First, only 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 (Figure 10). 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 into the corresponding bit in the round key table. However, such faults are an unlikely event. Faults injected at different rounds would not normally overlap, due to the non-linearity that is spreading the error very efficiently.

Injecting multiple faults in the MixColumn operation gives different results as the insertion affects the direct or the inverse routine. While the latter quickly becomes stable and distributes the error uniformly over the output word 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 may belong to a wider set of possible values, although it maintains some sort of parity

IEEE COMPUTER SOCIETY

property, i.e. the injection of an even (odd) number of faults results in an even (odd) number of output errors.

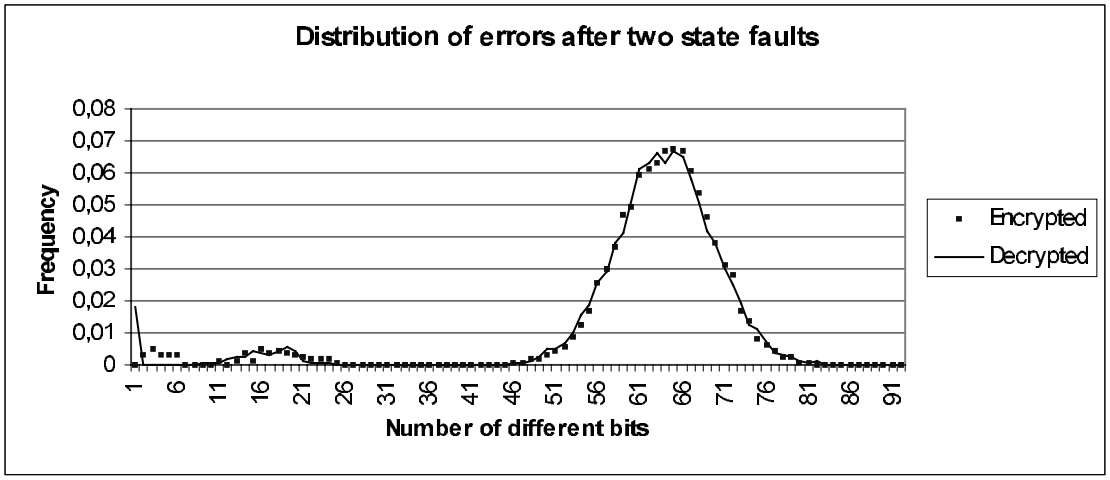

Figure 10: Effect of two fault injections in the state on the encrypted and decrypted output.

6. Techniques for detecting faults

A proposal for error detection in the data-path of AES was described in [10]. The goal of the proposed techniques was to prevent an attacker from attempts to break the cipher system by injection one or more incorrect bits. Our objective is different; we want to detect a fault in order to prevent the transmission and use of incorrect data.

The rather large number of bit errors, which we have observed for many different cases of injected faults, discourages the use of conventional error detecting codes. Still, some inner transformations might perhaps benefit from the usage of codes.

An architectural solution suggested in [1] is shown in Figure 11. It is based on the idea of performing a test decryption soon after encryption and then checking whether the original data block is obtained. If a decryption core is present in the implementation, the architecture overhead reduces to the cost of a comparator for two data blocks of 128 bits. The time necessary for detecting a fault is always equal to the time required to decrypt a data block, plus the time required for the comparison. Clearly this architecture is independent of the fault model.

Finer-grained error detection, able to inspect the internals of each one of the four transformations forming one round (SubByte, etc), would allow a faster reaction. This would avoid the execution of useless operations on already corrupted data, but could require a large overhead in terms of components, according to the implementation. Moreover, comparison should be done at the transformation level.

Based on the observations, suited points where such finer error detection can be performed seem to be the SubByte and the MixColumn transformations. SubByte is extremely non-linear, and thus the standard linear error detection codes are difficult to use (though not impossible). Methods like modular redundancy are possibly more suited. MixColumn is more linear (it is formed by a matrix multiplication, which is the linear part, followed by a constant addition, which is the non-linear part). Therefore, both error detection codes and modular redundancy techniques could be studied for MixColumn.

IEEE COMPUTER SOCIETY

Key scheduling deserves particular care since a fault in this part of the algorithm is not detected by the techniques used for detecting faults in the data-path. In [1] it was suggested to use a special key scheduling block, that could be called "inverse key scheduling", which allows to reconstruct the previous round key starting from the current one (see Figure 12). If the result of the inverse key schedule matches the previous key, the computation is correct and can proceed, otherwise a fault has been detected. Clearly this architecture is also independent of the fault model.

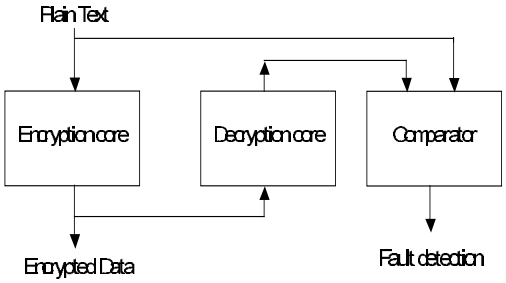

Figure 11: Fault detection at encryption level.

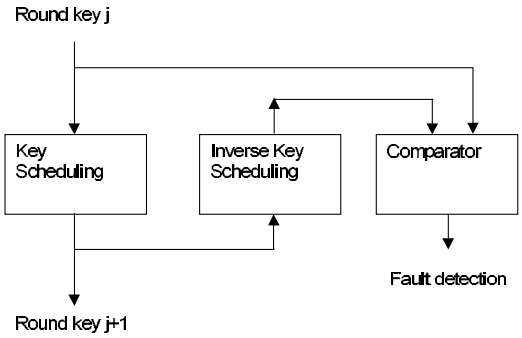

Figure 12: Logic scheme of the inverse key scheduling.

7. Conclusion

A detailed analysis of the behavior of the AES algorithm in the presence of faults has been carried out. This analysis considerably extends the one presented in [1]. Previous studies [10] have only considered the data-path, ignoring the key scheduling. The behavior of AES in the presence of faults seems to be relatively independent of the fault model. Several proposals for fault detection have been discussed. Future research directions include a more detailed study of fault detection and fault tolerance techniques covering the internal transformations of AES.

8. References

[1]. G. Bertoni, L. Breveglieri, I. Koren, V. Piuri, "Fault Detection In The Advanced Encryption Standard," to be presented in MPCS 2002, Ischia, Italy, 2002.

[2]. J. Daemen, V. Rijmen, "The Block Cipher Rijndael," Smart-Card Research and Applications, LNCS 1820, J.-J. Quisquater and B. Schneier, Eds., Springer-Verlag, 2000, pp. 288-296.

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

[4]. 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.

[5]. M. Akkar, C. Giraud, "An Implementation of DES and AES, secure against some attacks," Proceeding of CHES 2001, pp. 315-325.

[6]. M. McLoone, J. V. McCanny, "High Performance single-Chip FPGA Rijndael Algorithm implementations," Proceeding of CHES 2001, pp. 68-80.

[7]. V. Fischer, M. Drutarovsky, "Two Methods of Rijndael Implementation in reconfigurable Hardware," Proceeding of CHES 2001, pp. 81-96.

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

[9]. Rudra, P. K. Dubey, C. S. Jutla, V. Kumar, J. R. Rao, and P. Rohatgi, "Efficient Rijndael Encryption Implementation with composite Field Arithmetic," Proceeding of CHES 2001, pp. 175-188.

[10]. R. Karri, W. Kaijie, P. Mishra, K. Yongkook, "Fault-based side-channel cryptanalysis tolerant Rijndael symmetric block cipher architecture," Proceeding of Defect and Fault Tolerance in VLSI Systems, 2001 pp. 418 –426.

IEEE COMPUTER SOCIETY