Incorporating Error Detection and Online Reconfiguration into a Regular Architecture for the Advanced Encryption Standard

L. Breveglieri[1], I. Koren[2], P. Maistri[1]

[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

{brevegli,maistri}@elet.polimi.it, koren@ecs.umass.edu

Abstract

Fault injection based attacks on cryptographic devices aim at recovering the secret keys by inducing an error in the computation process. They are now considered a real threat and countermeasures against them must be taken. In this paper, we describe an extension to an existing AES architecture proposed by Mangard et al. [13], which provides error detection and fault tolerance by exploiting the high regularity of the architecture. The proposed design is capable of performing online error detection and reconfiguring internal data paths to protect against faults occurring in the computation process. We also describe how different redundancy levels provide protection against different numbers of errors. The presented design incorporating fault detection and tolerance has the same throughput as the base architecture but incurs a non-negligible area overhead. This overhead is about 40% *for the fault detection circuitry and* 134% *for the entire fault detection and tolerance (through reconfiguration). Although quite high, this overhead is still lower than for reference solutions such as duplication (providing detection) and triple modular redundancy (providing fault masking).*

1: Introduction

The AES algorithm is a symmetric block cipher standardized in 2001 by NIST [14] after allowing the research community to investigate possible weaknesses of the cipher. Since the standard has been established, many software and hardware implementations have been proposed. The software implementations have been targeting various platforms, from x86 architectures down to smartcard devices. The hardware solutions have been proposed for both reprogrammable devices, such as FPGAs, and custom devices (ASICs). The goals of these implementations are quite different. FPGA designs attempt to reduce the used gate count while keeping an acceptable throughput. This is usually achieved by sharing components between the encryption and decryption units, or by optimizing the mapping of the units onto the gate array [16]. Large throughput can be achieved by exploiting intensively inter-round and intra-round pipelining [18].

In contrast, ASICs are usually optimized for inclusion in embedded systems and consequently, the focus is on exploiting the maximum parallelism allowed by the algorithm and increasing the

Proceedings of the 2005 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'05)
0-7695-2464-8/05 $20.00 © 2005 IEEE

throughput, e.g., by unrolling the round iterations and pipelining the operations [8]. The higher throughput incurs a very large area requirement, see for instance [17].

The increasing complexity of cryptographic algorithms raises some new issues. First of all, it increases the likelihood of hardware failures, forcing us to check the validity of the result. Due to the diffusive behavior of symmetric ciphers, even a single bit error occurring in the middle of the computation can result in a completely different (and unusable, in the best case) result. Moreover, the occurrence of a computation error can be exploited by an attacker to easily recover the secret key with a very small number of attempts, each consisting of the pair made of the correct and the wrong output. When errors can be deliberately injected, this attack becomes a very powerful threat to cipher security. Errors can be induced in several ways, but the most effective way is by electromagnetic radiation (i.e., light or laser beam). The timing and location of the fault can be controlled precisely but the actual injected error can not be predicted and it is hence considered random. Some examples of fault attacks can be found in [4, 6] which address fault attacks against DES and RSA, respectively. Many attacks have also been proposed against AES after its standardization. These mostly affect the last computation rounds to exploit the controlled confusion features of the permutation layer (i.e., the *MixColumns* operation); see for instance [5, 7, 15].

Such fault injection attacks can be made ineffective by using some error detection scheme. An initial solution was proposed in [9] where error detection is achieved by exploiting the redundancy of functional units typical of encryption/decryption units. Later research has studied the application of error detecting codes to symmetric block ciphers [1, 10]. Simple codes like parity allow obtaining very high detection capabilities at a reasonable cost. Using parity code for fault detection in generic Substitution-Permutation Networks has also been discussed in [11].

So far, error codes were used only for detection purposes. However, it has been proven in [2] that the parity code at the byte level can also be used to locate the error even after several rounds have been completed. In this paper we show an extension to an existing architecture [13] incorporating our error detection scheme. Moreover, we further develop the architecture in order to provide fault tolerance through reconfiguration at computation time. As soon as an error is detected, the parity scheme allows to locate the faulty computation cell and a backup cell takes its place. Computation is then restarted from the beginning to provide the correct result.

The paper is organized as follows. In Section 2 we provide a brief overview of the AES encryption algorithm and the reference architecture. Section 3 describes the application of the error detection code and the structure of the fault tolerant architecture. The evaluation of the hardware costs and the performance impact are presented in Section 4. Section 5 concludes the paper.

2: The Advanced Encryption Standard

2.1: The AES algorithm

The Rijndael encryption algorithm [14] consists of three procedures, namely Encryption, Decryption (i.e., the inverse of Encryption) and Key Schedule. NIST imposed an input block size of 128 bits with the most common version using a 128-bit-long key, but longer keys are allowed. The ciphered output is 128-bit. The architecture we propose is limited to 128-bit-long keys; however, longer keys can be easily supported by modifying the Key Unit and the Control Unit.

Proceedings of the 2005 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'05)
0-7695-2464-8/05 $20.00 © 2005 IEEE

The encryption process consists of 10 iterative rounds executed after a pre-processing key-mixing phase, where the initial key is added (modulo 2) to the initial input. The intermediate result of the encryption process is stored in a 16-byte square matrix S called *state*. The encryption round is a chain of four byte-based transformations, both linear and non-linear, defined over the Galois Field $GF(2^8)$ (see Figure 1). These transformations are:

SubBytes: a byte-wise non-linear substitution, it can be computed on-the-fly or implemented by means of a lookup table;

ShiftRows: a byte rotation of the rows of the state S using an offset that depends on the row itself;

MixColumns: a linear algebraic transformation of each column of the state, over the Galois Field;

AddRoundKey: a bit-wise addition (modulo 2) of the current round key, provided by the key schedule, to the state S.

Each round key is also a 16-byte square matrix. The Key Schedule updates the matrix by using some of the encryption operations: in particular, each byte of the last row of the matrix is fed as input to the *SubBytes* operation. This result is later used to update each row of the matrix with a cascaded XOR-tree. Further details about AES can be found in [1]; Galois Fields are described in [14].

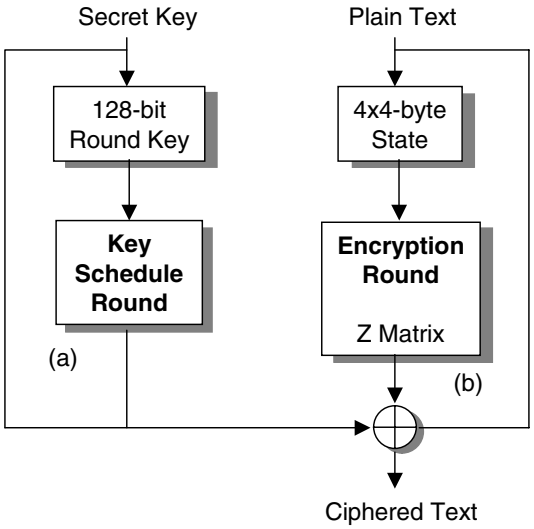

Figure 1. *Block diagram of Rijndael's AES: (a) Key Schedule, (b) Encryption.*

2.2: The reference architecture

The architecture we selected as a reference was presented in [13]. The authors described a highly regular architecture, which mimics the AES state: each byte of the state is computed and stored in a *data cell*, which communicates only with the cells in the same row or column (see Figure 2). Additionally, each cell has an extra input for the round key, not shown in the figure to make the

Proceedings of the 2005 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'05)
0-7695-2464-8/05 $20.00 © 2005 IEEE

drawing clearer. The first clock cycles are used to load the plain text into the circuit by feeding the input from the side. After the load is complete, the encryption process starts.

The actual computation process is slightly different for the two architectures. The baseline architecture has only 4 *SubBytes* units, which means that data is cycled vertically through each cell to enter the SBoxes and is updated at each cycle. The *ShiftRows* operation is computed by proper data routing at this stage. The *MixColumns* and *AddRoundKey* operations are computed in the next clock cycle. For this computation, each cell communicates with the other cells in the same column. Since the SBox implementation is pipelined, 6 clock cycles are needed to compute the *SubBytes* for the whole state and then complete the round. The Key Schedule updates the round key using the idle cycle of the SBoxes of the encryption datapath, thus saving some area.

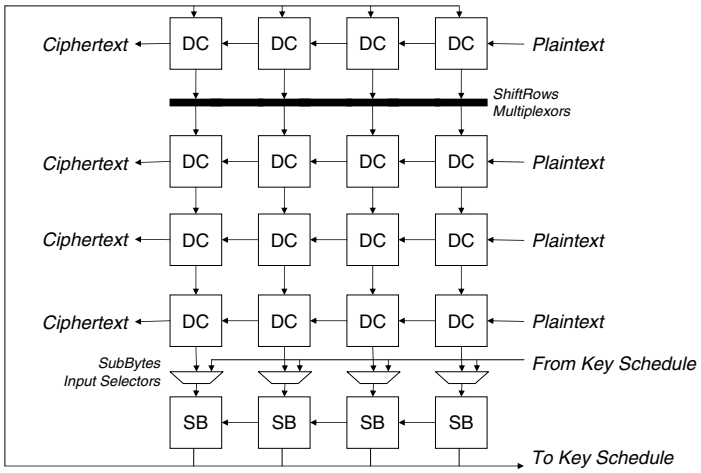

Figure 2. *Baseline reference architecture [13];* DC = *Data Cell,* SB = *SBox.*

The high-performance architecture adopts an SBox component for each data cell. Since the SBox is still pipelined, the SBoxes of the last row can also be used to update the round key. A single round is computed in only 3 clock cycles, at the expense of 16 global SBoxes. This architecture differs from the one presented above mainly in the number of SBoxes, since each data cell has its own. The fact that each data cell has its own SBox leads to some interesting consequences. First of all, there is no need to cycle the data vertically to compute *SubBytes*. Instead, the computation is performed locally by each data cell. Second, the *ShiftRows* operation must be implemented explicitly by connecting properly the data cells. Additional multiplexers are still needed to support both shifting directions, i.e., when decrypting data.

In the next section we discuss the changes we made to the architecture in order to support online error detection and reconfiguration.

Proceedings of the 2005 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'05)
0-7695-2464-8/05 $20.00 © 2005 IEEE

3: Online detection and reconfiguration

3.1: Detection through parity code

In order to provide error detection capabilities, we decided to adopt the parity code as described in [1]. The choice was motivated by the fact that both the code and the architecture are fine-tuned to work at byte level. The basic processing unit of the reference architecture is the data cell computing and storing a single byte. The granularity of the parity will therefore be one parity bit for each data cell.

Each data cell was upgraded to store the 9^{th} bit for parity and all the encryption operations were similarly extended to provide parity prediction during execution. The prediction for *ShiftRows* and *AddRoundKey* is trivial. The prediction for the *MixColumns* operation is made more complex by the need to support both the encryption and decryption procedures. The higher complexity of the prediction rule is due to the larger coefficients of the *Inverse MixColumns* matrix. Prediction for *SubBytes* follows the approach used in [3]. Due to the non-linearity of the operation, no simple rule exists and prediction is obtained by XOR'ing the input data bits, the input parity and the stored predicted parity (computed on the basis of a correct input). When the input is faulty, the error is easily propagated to the output (see Figure 3). This approach allows to implement the SBox by using composite fields, as in our case, and allows complete freedom in the implementation; the SBox parity table acts as the actual parity predictor, while the XOR'ing of all the input bits constitutes the error propagation component; i.e., if the input is faulty, the result is computed as if the data bits were correct, but the output parity is complemented in order to restore the inconsistency, which will signal the error at the next checkpoint.

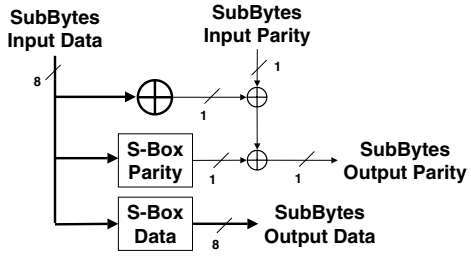

Figure 3. *Parity propagation in SubBytes.*

Parity prediction is implemented also in the Key Schedule component. However, since the operations used in key generation are those used also in the encryption datapath, we will skip the details. Moreover, the baseline reference architecture does not include any SBox in the Key Unit, relying instead on the SBoxes included in the Data Unit [13]. Thus, parity prediction is mainly parity propagation through XOR ports. Recall that our implementation supports only 128-bit keys, but longer keys can be easily supported as well, with a few changes in the control state machine.

In this architecture no explicit action is taken when an error is detected. Each single data cell outputs the predicted parity bit and an additional error bit, calculated as the exclusive OR between the predicted and the actual parity. The error bit can be used to signal the fault at the external interface or to take some more defensive action such as key memory deletion. Note however that this scheme targets round-level checkpoint frequency, which will be used in the next section. This

Proceedings of the 2005 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'05)
0-7695-2464-8/05 $20.00 © 2005 IEEE

means that there is no need to analyze the parity of the data input at each step. An erroneous input will be detected by the checkpoint unit anytime, since any single byte fault is propagated to the end of the process [2]. This approach greatly simplifies the global design.

3.2: Online reconfiguration

The parity error generated by each data cell can be used to route signals between correctly working components. This can be achieved by introducing a backup element and some multiplexers to route data to the correct path. Figure 4 shows how a backup data cell *BKP* can be used to substitute any data cell in the same row. When a cell signals an error in its result, its input is redirected towards the next data cell. Then, the input of the next data cell is redirected towards the end of the row, until the backup unit is fed with the last input value. This technique is well-known in fault tolerant array architectures [12]. For instance, suppose that there are 4 data cells in the row, named A, B, C, D and a backup cell E. If an error occurs in cell B, then its input is redirected to cell C; the original input of C is redirected to cell D, and finally, the original input of D is redirected to the backup cell E. The route to A is left untouched. In a similar way, the correct value must be selected at the output, in order to restore the correct data alignment: the lower layer of multiplexers, driven by the reconfiguration unit (which is an extension of the Control Unit), propagates the correct value. Using this approach, each row can be reconfigured to correct a single occurring error; this means that up to 4 errors can be corrected online, provided they occur in different rows. If 2 or more errors occur in the same row, reconfiguration is not possible and the device signals the inability to continue.

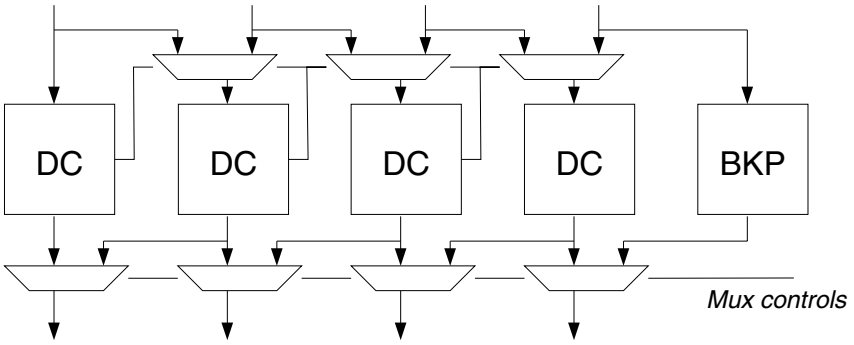

Figure 4. *Reconfigurable row with a backup element* (Arrows represent the data path; simple lines are the control signals).

The number of backup elements is not determined a priori. However, it should be emphasized that the data cells, although very similar one to one another, are not identical. In fact, the coefficients of the *MixColumns* operations are aligned according to the specific row (this is due to the property of the Galois Field $GF((2^8)^4)$ and the reducing polynomial chosen). For this reason, and to keep the design simple, we did not use a single backup cell for the whole data unit, although this is possible. Instead, we added a backup cell to each row. Since all the cells in the same row are similar (they have the same *MixColumns* coefficients), data can be easily moved. Other solutions are still possible: a single backup cell for the whole *Data Unit* as mentioned above, or a whole backup column, managed with the same policies of the backup cell described in this paper. In

Proceedings of the 2005 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'05)
0-7695-2464-8/05 $20.00 © 2005 IEEE

the latter, when a cell is marked as faulty, the whole column is disabled and its input routed to the adjacent column, and so on until the backup column is used. This reduces the number of multiplexers required for proper routing of each cell input/output and thus simplifies the design. Higher levels of redundancy can be implemented as well (e.g., a backup cell for each pair of data cells), but the area overhead is likely to increase drastically.

4: Experimental results

The basic architecture was described in VHDL and then extended with error detection capabilities first, and backup cells later. The architecture was synthesized three times, with target frequencies of 200, 250 and 333 MHz. However, the latter was too restrictive and could not be met. Here we report the results for the first two with target clock latencies of 5 and 4 ns, see Table 1. The Table also includes our previous solution [3] which is a round-iterated architecture using a single-stage pipeline and able to compute a complete encryption round in a single clock cycle. The new architecture is more globally efficient (see column AT of Table 1) and can perform both encryption and decryption.

We do not compare our results to the original solution in [13] since the technology libraries are different. Instead, we want to investigate the overheads introduced by the redundant components and provide a complete evaluation of the additional area and time requirements. Moreover, we compare the solution proposed in this paper to our previous architecture presented in [3], which supported only error detection.

We can analyze these results from different aspects. First of all, we can simply evaluate the overheads with respect to the bare solution with no extensions. We want to highlight the fact that there is no additional latency due to the code prediction logic or to the control logic used for reconfiguration. Thanks to pipelining, the clock period is preserved, but on the other hand, the increase in area is significant. Comparing to the error detecting version in [3], we note that the overhead is much higher (about 40% versus only 18%). However, the previous design suffered from a non negligible increase in latency, which does not occur in the current solution.

Table 1. *Synthesis results and comparison.*

Version	Area μm^2	*overhead*	Latency *ns*	*overhead*	AT	*overhead*
DFT '04 - 1 stage pipeline						
Base	233,100	–	8.9	–	2,074,590	–
Detection	276,500	+18%	12.0	+35%	3,318,000	+60%
DFT '05 - Target clock 5ns - 3 stage pipeline						
Base	147,200	–	4.6	–	677,120	–
Detection	212,500	+44%	4.6	+0%	977,500	+44%
Tolerance	385,600	+162%	4.6	+0%	1,773,760	+162%
DFT '05 - Target clock 4ns - 3 stage pipeline						
Base	190,800	–	3.6	–	686,900	–
Detection	266,700	+39%	3.6	+0%	960,120	+39%
Tolerance	446,500	+134%	3.6	+0%	1,607,400	+134%

Proceedings of the 2005 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'05)
0-7695-2464-8/05 $20.00 © 2005 IEEE

5: Conclusions

Attacks on cryptographic devices based on fault injection can now be considered a real threat. In this paper, we have proposed an extension to an existing AES architecture, in order to provide error detection and fault tolerance to protect against malicious fault injections. Thanks to pipelining, the system throughput is unchanged, but the area overhead is significant (e.g., between 40% and 134%). This overhead is still lower than conventional solutions like duplication and Triple Modular Redundancy (TMR). Moreover, our fault tolerant solution is able to sustain up to 4 independent errors, while TMR can tolerate only one fault.

Acknowledgment: 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 Defense Advanced Projects Agency, AFRL, or the US Government.

References

[1] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri and V. Piuri, "Error Detection Procedures for a Hardware Implementation of the Advanced Encryption Standard," *IEEE Transactions on Computers*, Special Issue on Cryptographic Hardware and Embedded Software, pp. 492-505, April 2003.

[2] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri and V. Piuri, "Detecting and locating faults in VLSI implementations of the Advanced Encryption Standard," *Proc. of the 2003 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pp. 105-113, 2003.

[3] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, "An efficient hardware-based fault diagnosis scheme for AES: performances and cost," *Proc. of the 2004 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pp. 130-138, Oct. 2004.

[4] E. Biham, A. Shamir, "Differential Fault Analysis of Secret Key Cryptosystems," Technical Report in Technion - Computer Science Department, 1997.

[5] J. Blömer, J.-P. Seifert, "Fault Based Cryptanalysis of the Advanced Encryption Standard (AES)," Financial Cryptography, *Lecture Notes in Computer Science*, vol. 2742, pp. 162-181, 2003.

[6] D. Boneh, R. DeMillo, R. Lipton, "On the Importance of Eliminating Errors in Cryptographic Computations," *Journal of Cryptology*, vol. 14, pp. 101-119, 2001.

[7] C. Giraud, "DFA on AES," http://eprint.iacr.org/2003/008.ps.gz

[8] A. Hodjat, I. Verbauwhede, "Minimum area cost for a 30 to 70 Gbits/s AES processor," *VLSI, 2004. Proceedings. IEEE Computer society Annual Symposium on*, pp. 83-88, 2004.

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

[10] R. Karri, G. Kuznetsov, M. Goessel, "Parity-based concurrent error detection in symmetric block ciphers," *Proceedings of International Test Conference 2003 - ITC 2003*, Volume 1, ISSN 1089-3539, pp. 919 - 926, 2003.

Proceedings of the 2005 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'05)
0-7695-2464-8/05 $20.00 © 2005 IEEE

[11] R. Karri, G. Kuznetsov, M. Goessel, "Parity-Based Concurrent Error Detection of Substitution-Permutation Network Block Ciphers," Cryptographic Hardware and Embedded Systems - CHES 2003, *Lecture Notes in Computer Science*, vol. 2779, pp. 319-333, Springer-Verlag, 2003.

[12] S. Y. Kung, *VLSI Array Processors*, Prentice-Hall, ISBN 013942749X, 1988.

[13] S. Mangard, M. Aigner, S. Dominikus, "A highly regular and scalable AES hardware architecture," IEEE Transactions on Computers, Volume 52, Issue 4, pp. 483-491, April 2003.

[14] National Institute of Standards and Technologies, "Announcing the Advanced Encryption Standard (AES)," *Federal Information Processing Standards Publication*, n. 197, November 26, 2001.

[15] G. Piret, J.-J. Quisquater, "A Differential Fault Attack Technique against SPN Structures, with Application to the AES and Khazad," Cryptographic Hardware and Embedded Systems - CHES 2003, *Lecture Notes in Computer Science*, vol. 2779, Springer-Verlag, pp. 77-88, 2003.

[16] F.-X. Standaert, G. Rouvroy, J.-J. Quisquater, J.-D. Legat, "Efficient Implementation of Rijndael Encryption in Reconfigurable Hardware: Improvements and Design Tradeoffs," Cryptographic Hardware and Embedded Systems - CHES 2003, *Lecture Notes in Computer Science*, vol. 2779, Springer-Verlag, pp. 334-350, 2003.

[17] I. Verbauwhede, P. Schaumont, H. Kuo, "Design and performance testing of a 2.29-GB/s Rijndael processor," *IEEE Journal of Solid-State Circuits*, Vol. 38, Issue 3, pp. 569-572, 2003.

[18] X. Zhang, K.K. Parhi, "High-speed VLSI architectures for the AES algorithm," Very Large Scale Integration (VLSI) Systems, IEEE Transactions on," vol. 12, Issue 9, pp. 957-967, 2004.

Proceedings of the 2005 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'05)
0-7695-2464-8/05 $20.00 © 2005 IEEE