# Low Cost Concurrent Error Detection for the Advanced Encryption Standard

Kaijie Wu  Ramesh Karri
Electrical and Computer Engineering Dept
Polytechnic University
6 Metrotech Center
Brooklyn, NY 11201

Grigori Kuznetsov  Michael Goessel
University of Potsdam
Institute of Computer Science
Fault Tolerant Computing Group
D-14439 Potsdam, Germany

## Abstract

*We present a new low-cost concurrent checking method for the Advanced Encryption Standard (AES) encryption algorithm. In this method, the parity of the 128-bit input is determined and modified step-by-step into the parity of the 128-bit output according to the processing steps of the AES encryption. For the parity-preserving AES steps Shift-Rows and Mix-Column no parity modifications are necessary. The modified parity is compared in any round with the actual parity of the outputs of the round. To obtain the hardware costs we implemented this method on a Xilinx Virtex 1000 FPGA. For this implementation, the hardware overhead is about 8% and the additional time delay is about 5%. The method detects technical faults and deliberately injected faults during normal operation.*

## 1. Introduction

In 2001 the 128-bit Advanced Encryption Standard (AES) [1] replaced the 64-bit Data Encryption Standard (DES) as the symmetric encryption algorithm standard. Since then AES has become the most popular encryption algorithm to be implemented on smart cards and other devices. Because of the rapidly shrinking dimensions in VLSI, transient faults occur and will occur in the near future in increasing numbers. These transient faults affect the memory as well as the combinational parts of a circuit and can only be detected using concurrent checking. This is especially true for sensitive devices such as cryptographic chips. Hence, concurrent checking for cryptographic chips is growing in importance. Since cryptographic chips are a consumer product produced in large quantities, cheap solutions for concurrent checking are needed. Concurrent checking for cryptographic chips has also a great potential for detecting (deliberate) fault injection attacks where faults are injected into a cryptographic chip to break the key [2][3][4][5].

Until now the following concurrent checking methods for cryptographic algorithms are known. In [6] a Register Transfer Level concurrent checking approach for AES and other symmetric block ciphers that exploits the inverse relationship between the encryption and decryption at the algorithm level, round level and individual operation level was developed. This technique has an area overhead of 21% at the algorithm level, 18.9% at the round level and 38.08% at operation level. Similarly, the time overhead is 61.15%, 26.55% and 23.56% respectively. In [9] this inverse-relationship technique was extended to AES round key generation. A drawback of this approach is that it assumes that the cipher device operates in a half-duplex mode (i.e. either encryption or decryption but not both are simultaneously active).

In [10] the first parity-based method of concurrent checking for the AES encryption algorithm was presented. This technique has relatively high hardware overhead. The technique adds one additional parity bit per byte resulting in 16 additional bits for the 128-bit data stream. Each of the sixteen 8-bit×8-bit AES s-boxes is modified into 9-bit×9-bit s-boxes more than duplicating the hardware for implementing the s-boxes. In addition, this technique adds one additional parity bit per byte to the outputs of the Mix-Column operation because Mix-Column does not preserve parity of its inputs at the byte-level.

To reduce the necessary hardware overhead for concurrent checking of the AES encryption algorithm we propose a new approach for low-cost parity checking. This method is based on a general method of concurrent checking for Substitution Permutation Networks (SPN) [7][8]. In this method, the input parity of an SPN network is modified according to the processing steps of the SPN network into the output parity and compared with the output parity of every round. In this paper, we adapt this general approach to develop a low-cost method for concurrent checking of the 128-bit AES encryption algorithm. First, we determine the parity of the 128-bit input using a parity tree of exclusive-or gates. In AES encryption each of the 16 input bytes are processed by an 8-bit×8-bit nonlinear s-box that implements a polynomial inversion in $GF(2^8)$ and a linear affine transformation. For concurrent error detection, we add one additional binary output to each of the 16 s-boxes. This additional s-box output computes the exclusive-or of parity of each 8-bit input and the parity of the corresponding 8-bit output.

0-7803-8580-2/04 $20.00 Copyright 2004 IEEE

Each of the modified s-boxes are 8-bit×9-bit in size. The additional 1-bit output of the 16 s-boxes are used to modify the input parity for this nonlinear Sub-Bytes processing step. Since the Shift-Rows operation implements a permutation, it does not change the parity from its input to its output. This operation can be implemented by proper wiring in hardware. Surprisingly, the Mix-Column operation for every group of 32 bits does not change the parity from its inputs to its outputs as well. Finally, component-wise exclusive-or of the 128-bit round key needs a parity modification by a single pre-computed parity bit of the round key. Since the output of a round is the input to the next round, the output parity of the outputs of a round can be computed using the same hardware which was used for computing the input parity of the previous round.

The paper is organized as follows. In section **Error! Reference source not found.** we will briefly recapitulate the AES encryption. In section 0 we will describe the proposed low-cost method of concurrent checking. The error detection capability of the proposed method is discussed in section **Error! Reference source not found.**. To obtain the area overhead and the additional time delay the method was implemented on an FPGA. The results of this basic implementation and additional optimizations specific to an FPGA-implementation are presented in section **Error! Reference source not found.**. Conclusions are reported in section **Error! Reference source not found.**.

## 2. Advanced Encryption Standard

AES [1] is an iterative block cipher with a variable block length. In this paper, we will consider a block length of 128 bits. The concurrent checking described in this paper is applicable to other block lengths of AES as well. The AES algorithm encrypts a 128-bit input plain text into a 128-bit output cipher text using a user key using 10 almost identical iterative rounds. The 128-bit (or 16-byte) input and the 128-bit (or 16-byte) intermediate results are organized as a 4×4 matrix of bytes called the state **X**.

$$X = \begin{bmatrix} x_0 & x_4 & x_8 & x_{12} \\ x_1 & x_5 & x_9 & x_{13} \\ x_2 & x_6 & x_{10} & x_{14} \\ x_3 & x_7 & x_{11} & x_{15} \end{bmatrix}$$

The four four-byte groups ($x_0$, $x_1$, $x_2$, $x_3$), ($x_4$, $x_5$, $x_6$, $x_7$), ($x_8$, $x_9$, $x_{10}$, $x_{11}$) and ($x_{12}$, $x_{13}$, $x_{14}$, $x_{15}$) form the first, the second, the third and the fourth columns respectively of the state matrix A. The four four-byte groups ($x_0$, $x_4$, $x_8$, $x_{12}$), ($x_1$, $x_5$, $x_9$, $x_{13}$), ($x_2$, $x_6$, $x_{10}$, $x_{14}$) and ($x_3$, $x_7$, $x_{11}$, $x_{15}$) form the four rows of the state (matrix) **X**.

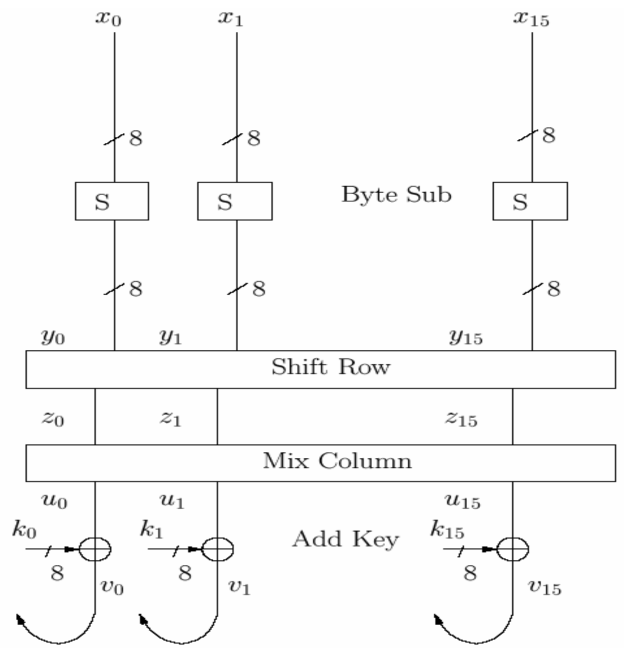

Figure 1: AES round without concurrent checking

One round of the AES encryption algorithm (without concurrent checking) is shown in Figure 1. It consists of the operations Sub-Bytes, Shift-Rows, Mix-Column, and Add-key. In the last round the Mix-Column operation is not used. We summarize these operations next.

### 2.1 Sub-Bytes:

All bytes are processed separately. For every byte not equal to $\underline{0}$=(0,0,0,0,0,0,0,0) first the inverse in $GF(2^8)$ is determined. $m(x)=x^8+x^4+x+1$ is used as the modular polynomial for $GF(2^8)$. The byte $\underline{0}$ is mapped to $\underline{0}$. Then a linear affine transformation is applied. Very often Sub-Bytes is implemented using 16 copies of an 8-bit×8-bit ROM. The result state is represented as **Y**:

$$Y = \begin{bmatrix} y_0 & y_4 & y_8 & y_{12} \\ y_1 & y_5 & y_9 & y_{13} \\ y_2 & y_6 & y_{10} & y_{14} \\ y_3 & y_7 & y_{11} & y_{15} \end{bmatrix}$$

### 2.2 Shift-Rows:

The rows of the state are shifted cyclically byte-wise using a different offset for each row. Row 0 is not shifted, row 1 is cyclically shifted left 1 byte, row 2 is cyclically shifted left by 2 bytes and row 3 is cyclically shifted left 3 bytes. The result state is represented as **Z**:

$$Z = \begin{bmatrix} z_0 & z_4 & z_8 & z_{12} \\ z_1 & z_5 & z_9 & z_{13} \\ z_2 & z_6 & z_{10} & z_{14} \\ z_3 & z_7 & z_{11} & z_{15} \end{bmatrix} = \begin{bmatrix} y_0 & y_4 & y_8 & y_{12} \\ y_5 & y_9 & y_{13} & y_1 \\ y_{10} & y_{14} & y_2 & y_6 \\ y_{15} & y_3 & y_7 & y_{11} \end{bmatrix}$$

### 2.3 Mix-Column:

The elements of the columns of the state are considered as the coefficients of polynomials of maximal degree 3. The coefficients are considered as elements of $GF(2^8)$. These polynomials are multiplied modulo the polynomial $x^4+1$ with a fixed polynomial $c(x) = (03)x^3+(01)x^2+(01)x+(02)$. The coefficients of this polynomial given in hexadecimal representation are also elements of $GF(2^8)$. Thus for instance the coefficient (03) in hexadecimal representation is in binary representation $(000000011)_2$ or x+1 in polynomial representation in $GF(2^8)$. The Mix-Column operation on a column $z^T=[z_0, z_1, z_2, z_3]^T$ of the state into the column $u^T=[u_0, u_1, u_2, u_3]^T$ can be formally described by Equation 1 where the constant elements of the matrix **C** and of the vectors $z^T$ and $u^T$ as well as the multiplication and the addition are in $GF(2^8)$. The polynomial $x=x^8+x^4+x+1$ is used as the modular polynomial. The elements of the matrix **C** are 01, 02 and 03.

$$\begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{bmatrix} \qquad (1)$$

Multiplication of an 8-bit coefficient $z_i = z_{i,0},\ldots\ldots,z_{i,7}$ by (01) does not change $a_i$ (i.e., $(01)\times z_i= z_i$). Multiplication modulo $x^8+x^4+x+1$ by constant (02) (or by the polynomial x) first results in a left-shift of $a_i$. If the MSB $z_{i,7}$ of $z_i$ is 1 then after this left-shift the MSB is deleted and $(00011011)_2$ is added which, in $GF(2^8)$ corresponds to a bitwise exclusive-or operation.

$(02)\times z_i = (02)\times(z_{i,7},z_{i,6},z_{i,5},z_{i,4},z_{i,3},z_{i,2},z_{i,1},z_{i,0}) = (z_{i,6},z_{i,5},z_{i,4},z_{i,3},z_{i,2},z_{i,1},z_{i,0},0)\oplus(0,0,0,z_{i,7},z_{i,7},0,z_{i,7},z_{i,7})$.

Multiplication by (02) can be implemented three exclusive-or gates. Since $(03) = (02)\oplus(01)$, $(03)\times z_i$ by can be implemented as an exclusive-or of $z_i$ and $(02)\times z_i$ (i.e., $(03)\times z_i=z_i\oplus(02)\times z_i$). Overall, the Mix-Column operation can be implemented by a simple linear network of exclusive-or elements.

### 2.4 Add-key:

Add-key operation is a bit-wise exclusive-or of the 128-bit round key with the 128-bit state.

## 3. Low-Cost Concurrent Checking of AES

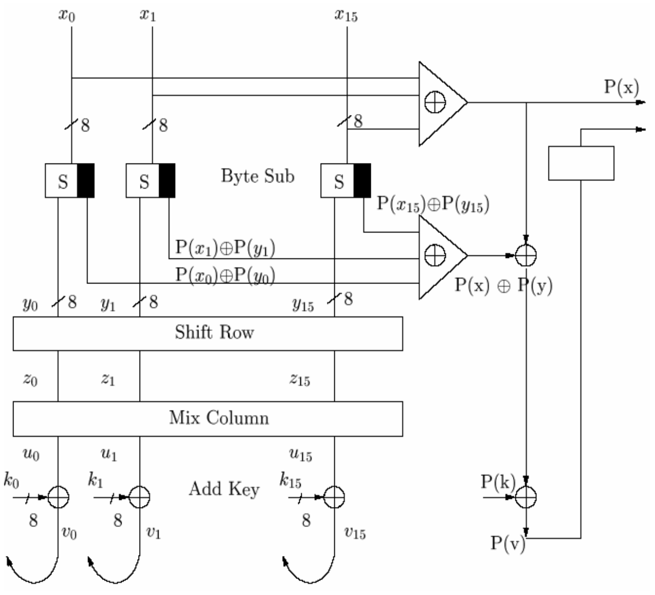

Figure 2: AES round with concurrent checking

We will now describe step-by-step the parity code based low-cost checking for AES encryption. A single round of the AES encryption with concurrent checking is shown in Figure 2. For concurrent checking the parity of the inputs of a round is determined, modified according to the processing steps of the AES encryption into the parity of the outputs and compared with the actual parity of the outputs of this round. We will explain the modifications necessary to the input parity for every step of AES encryption.

**Step 1: Computing the input parity:** Parity of the 128-bit input is determined by a tree of exclusive-or gates. As will become clear later it is useful to compute the parities of each of the 16 bytes first and then combine these 16 parity bits using a tree of exclusive-or gates to get the parity bit for the 128-bit input.

**Step 2: Parity modification according to Sub-Bytes:** Without error detection all 16 bytes of the 128-bit input are processed by an 8-bit input, 8-bit output nonlinear function called s-box. An s-box implements a polynomial inversion in $GF(2^8)$ followed by a linear affine transformation. An s-box can be either implemented by an 8-input, 8-output ROM or, as combinational logic.

For error detection in an s-box we add an additional binary parity output which implements exclusive-or of the parity of all 8-bit inputs with the parity of the corresponding 8-bit outputs. The values for this additional output can be determined from the truth table of the original s-box. The modified s-box has an 8-bit input and a 9-bit output. In Figure 2 this additional parity output is shown as a thick box appended to the right hand side of an s-box. The parity modification for the complete 128-bit operation Sub-Bytes is determined as the exclusive-or of the 16 additional parity outputs of the sixteen s-boxes using a 16-input one-output tree of exclusive-or gates. To modify the input parity according to the Sub-Bytes processing step the output of this parity tree is exclusive-ored with the input parity.

The modified s-boxes can be either implemented by ROMs with 8-bit inputs and 9-bit outputs or as a combinational logic circuit. If we implement the 8 functional outputs and the parity output of an s-box as combinational logic, a two-level implementation is good for error detection. In such a two-level realization every single internal fault in the s-box is immediately detected by an erroneous parity. The overhead for implementing the additional parity modification bit of an s-box is ~12.5% of the area for the basic s-box.

**Step 3: No parity modification according to Shift-Rows:** Shift-Rows operation is a permutation of the bytes of the state and obviously does not change the parity. Therefore the parity is not modified in this step.

**Step 4: No parity modification according to Mix-Column:** The Mix-Column operation is described by equation (1). Because of the simple structure of the coefficients in the matrix **C** this operation can be easily implemented as a linear in *GF(2)* combinational circuit.

Surprisingly, the Mix-Column operation does not alter the parity from its input to its outputs when the 32-bit columns of the state are considered. Therefore if we add a single parity bit for every column (of 32 bits) of the state or a single parity bit for the entire 128-bit state, these parity bits are not modified by the Mix-Column operation.

To prove this statement, consider the first column of the input to Mix-Column operation **Z** ($z_0$, $z_1$, $z_2$, $z_3$) and the first column of the output of Mix-Column operation **U** ($u_0$, $u_1$, $u_2$, $u_3$). Let $P(z_i)$ and $P(u_i)$ are parities of the bytes. Since Mix-Column operation is defined by equation 1, we have:

$u_0 = (02)z_0 + (03)z_1 + (01)z_2 + (01)z_3$

$u_1 = (01)z_0 + (02)z_1 + (03)z_2 + (01)z_3$

$u_2 = (01)z_0 + (01)z_1 + (02)z_2 + (03)z_3$

$u_3 = (03)z_0 + (01)z_1 + (01)z_2 + (02)z_3$

The parity of the 32-bit output column generated by Mix-Column is

$P(u_0)\oplus P(u_1)\oplus P(u_2)\oplus P(u_3) = P(u_0+u_1+u_2+u_3)$

$= P[(02)z_0+(03)z_1+(01)z_2+(01)z_3+(01)z_0+(02)z_1+(03)z_2+(01)z_3+$

$(01)z_0+(01)z_1+(02)z_2+(03)z_3+(03)z_0+(01)z_1+(01)z_2+(02)z_3]$.

Substituting $(01)z_i+(02)z_i+(03)z_i=0$ and $(03)z_i+(03)z_i=0$ for i = 0, 1, 2, 3,

$P(u_0)\oplus P(u_1)\oplus P(u_2)\oplus P(u_3) = P(z_0)\oplus P(z_1)\oplus P(z_2)\oplus P(z_3)$.

This proves that the parity of a column consisting of four bytes of the state is not changed by the Mix-Column operation. In Figure 2 we use a single parity bit for all 128-bits for concurrent checking; since this is not modified by the Mix-Column operation it does not entail CED area overhead.

In [10] a parity bit is used for every byte of the state. However, since the Mix-Column operation alters the parity of every byte of the state, the input parity of the input bytes to the Mix-Column has to be modified resulting in an area overhead.

**Step 5: Parity modification according to Add-Key:** Since a 128-bit round key is component-wise exclusive-ored with the 128-bit state output of Mix-Column operation, the input parity has to be modified by the parity $P(k) = k_0\oplus k_1\oplus\ldots\oplus k_{127}$ of the round key as shown in Figure 2. This parity for all round keys is pre-computed during round key generation. This parity modification step requires a 1-bit exclusive or gate.

**Step 6: Output parity checking:** The parity of the actual outputs of the round has to be compared with the modified input parity of the round. For this purpose, the modified input parity is stored in a 1-bit register. Since the outputs of a round are connected to the inputs of the round, the input parity of a round (computed using exclusive-or tree at the inputs of the round in step 1) is also the output parity of the previous round. Comparing this parity with the modified input parity of the previous round stored in the 1-bit register detects a possible error.

Overall, for concurrent checking of AES encryption the input parity is determined and this input parity is modified only for Sub-Bytes operation by additional outputs of the 16 s-boxes and by 1-bit pre-computed parity of the round key.

## 4. CED Capability

In AES and other encryption algorithms a single error bit in one location spreads within a few rounds to a large number of (even or odd) erroneous bits [11]. This is confirmed by simulation experiments in [10]. Because of this property traditional parity prediction for encryption algorithms is not easy. For the proposed parity modification the situation is different. Using the proposed method of step-by-step parity modification a local fault within a processing step which is in principle detectable by parity checking at the outputs of this processing step will also be detected by comparing the modified parity with the actual parity of the outputs of the round.

**Theorem:** If a fault in a processing step in a round (Sub-Bytes, Shift-Rows, Mix-Column, Add-Key) is locally detectable by parity at the outputs of this processing step then the fault will be detected by comparing the modified input parity with the actual output parity of the round.

**Proof:** Let us refer to Figure 3. Let x be the correct input to the considered round and let y, z, u, and v be the correct outputs of the processing steps Sub-Bytes, Shift-Rows, Mix-Column and Add-Key respectively. Input parity P(x) is modified by $\Delta$=P(x)$\oplus$P(y) into P(y) ($\Delta\oplus$ P(x)) according to the Sub-Bytes operation. Since Shift-Rows and Mix-Column do not alter parity, P(y) = P(z) = P(u). Finally, P(v)=P(u) $\oplus$ P(k) and the modified parity $P_M$, is equal to the parity P(v) of the outputs v of the Add-Key operation.

Let us see how a stuck-at fault at the inputs of Sub-Bytes operation can be detected. Let us replace the correct input x by an erroneous value x', x≠x'. If Sub-Bytes is implemented using ROMs the considered error corresponds to an address error of the ROM. If Sub-Bytes is implemented as combinational logic the considered error corresponds to a stuck-at fault at an input-line of a gate of the first level of the implementation. For this case the outputs of Sub-Bytes, Shift-Rows, Mix-Column and Add-Key in Figure 3 are denoted by y', z', u' and v' respectively. The input parity P(x) is now modified by $\Delta$ = P(x') $\oplus$ P(y') into P(x) $\oplus$ P(x') $\oplus$ P(y') and the modified parity $P_M$ is P(x) $\oplus$ P(x') $\oplus$ P(y') $\oplus$ P(k). For the parity P(v') of the outputs of this round we have P(v') = P(u') $\oplus$ P(k) = P(z') $\oplus$ P(k) = P(y') $\oplus$ P(k) and P(v') ≠ $P_M$ if P(x') ≠ P(x).

Thus the input error (e.g. an address error of a ROM) is detected by comparing the modified parity and the actual parity of the outputs of the round if P(x') ≠ P(x).

As a second example, let us consider a technical fault $f^1$ within the processing step Sub-Bytes. Let $y^1$, $z^1$, $u^1$, $v^1$ be the outputs of the processing steps Sub-Bytes, Shift-Rows, Mix-Column and Add-Key respectively due to this technical fault as shown in Figure 3. In this case we have $P_M$ = P(x) $\oplus$ P(x) $\oplus$ P(y) $\oplus$ P(k) = P(y) $\oplus$ P(k) and P($v^1$) = P($u^1$) $\oplus$ P(k) = P($z^1$) $\oplus$ P(k) = P($y^1$) $\oplus$ P(k) ≠ $P_M$ and the fault is detected for P(y) ≠ P($y^1$).

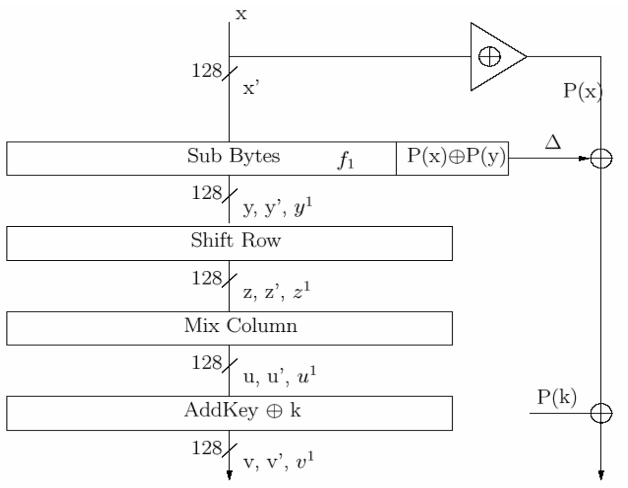

Figure 3: CED capability of AES with concurrent checking

In a similar way faults in the processing steps Shift-Rows, Mix-Column and Add-Key are detected. If we consider single stuck-at faults as the technical faults in the processing steps or injected single bit faults then the number of erroneous bits at the outputs of these processing steps depends on the concrete implementation. According to [12] in random logic the probability of 1-bit, 2-bit, 3-bit and 4-bit errors before (after) optimization is 88.8(77.9)%, 7.7(12.8)%, 1.9(4.7)% and 1.1(2.4)% respectively. For the different AES processing steps special designs with good error detection properties are possible.

At the gate level different implementations of the S-boxes are considered. If all the bits of the S-boxes and also the parity bits are separately implemented the necessary area is 79% of the not optimized circuit. 100% of the errors are detected. If the bits of the s-boxes are jointly optimized and if the parity bit is separately implemented the area is only 35% of the area of the not optimized circuit. But in this case many two-bit errors are generated within the S-boxes by single stuck-at faults and only 48-53% of the errors due to these single stuck-at faults in the s-boxes are immediately detected. The situation for the detection of the single stuck-at faults is quite different. 98.7% of the single stuck-at faults are detected at least with some latency. All the other parts of the implementation remain unchanged

Then all possible faults are detectable by parity checking and all faults are detected by comparing the modified parity with the actual parity of the outputs of the round. The operation Shift-Rows is a byte-wise permutation and this operation can be implemented by proper wiring in hardware. All single faults will result in single errors. The Mix-Column operation (composed of exclusive-or operations) can be implemented in which all single stuck-at faults and all injected single faults result in an odd number of erroneous bits at its outputs which are detectable by parity checking. This is also confirmed by the simulation experiments of [10]. Add-Key is a component-wise exclusive-or of 128 bits. All single stuck-at faults and all single injected faults will result in one-bit errors which are once again detectable by parity checking. Stuck-at faults at the outputs of the parity trees are detected possibly with a small latency. As in the FPGA-based implementation a separate parity line for every block of 32 bits can be used to increase the fault coverage for multiple stuck-at faults.

## 5. FPGA Implementation

We used a Xilinx Virtex 1000 FPGA device to prototype the described CED method. We used Synplify Pro for synthesis and Xilinx ISE for place and route steps. In this design, we implemented the sixteen s-boxes using sixteen identical 8-bit×9-bit ROMs. Each ROM contains 256 bytes of s-box data and 256 bits for the corresponding parities. The additional parity column translates into an overhead of sixteen 256x1 ROMs. The input and output parities are generated using a 128-bit parity tree using 46 Look Up Tables (LUT). Each LUT in the target FPGA can implement a 4-input-1-output logic.

Table 1: Area and time overhead for AES with CED

| | Non-CED | CED | Overhead |
|---|---|---|---|
| # of LUT | 3629 | 3899 | 7.4% |
| # of register bits | 823 | 830 | 1% |
| Clock period (ns) | 17.742 | 18.883 | 6.4% |

Table 1 summarizes the area and time numbers for the basic AES and AES with CED. The number of LUTs reports the combinational logic used in our implementation while the number of register bits (excluding register I/O) reports the sequential logic. The clock period is reported after place and route and includes LUT cell delays and routing delays. According to this table we have less than 8% hardware and less than 7% time overhead.

Xilinx Virtex FPGAs have a large amount of block RAMs. Efficient implementations for the AES without error detection using block RAMS Xilinx FPGAs are described in [13][14]. A block RAM can be configured as a dual-port ROM with two 8-bit input ports and two 9-bit output ports. Such a dual-port ROM functions as two independent single-port ROMs which share identical ROM contents. The area of a dual-port ROM is the same as the area of a single-input-output ROM. Since the sixteen AES s-boxes are identical to each other, we propose to use eight dual-port ROMs yielding a fifty percent reduction compared to the basic CED implementations. In turn, the dual-port ROM based implementation will reduce the number of columns used to compute the parity term from 16 to 8.

Dual-port ROMs based implementation will compromise the error detection capability of the 1-bit parity based concurrent checking. When a fault is introduced into a dual-port ROM, this faulty content may appear twice in the output if the values on the two input ports are the same. The two faulty outputs cancel each other and do not change the parity of the 128-bit output. To detect such faults, we need to use at least two parity bits (one parity bit for 64 bits). We propose to use four parity bits with one parity bit for every 32 bits. In order to detect all faults in the dual-port ROMs, we need to carefully identify the 32-bit input whose parity is computed, modified step-by-step and compared with the corresponding 32-bit output.

In Figure 1, **X** is the input to the AES round and **Y**, **Z**, **U**, **V**, are the outputs of Sub-Bytes, Shift-Row, Mix-Column, and Add-Key operations respectively. Let us start with the AES round output **V**. The parity of the 32-bit first column ($v_0$, $v_1$, $v_2$, $v_3$) of **V** is determined by the parity of the 32-bit first column ($u_0$, $u_1$, $u_2$, $u_3$) of **U**. This in turn is determined by the parity of the first column ($z_0$, $z_1$, $z_2$, $z_3$) of **Z**. Since the Shift-Row operation shifts the positions of the bytes in a column from its inputs to its outputs, the parity of the first column of **Z** ($z_0$, $z_1$, $z_2$, $z_3$) is determined by the parity of the 32 bits ($y_0$, $y_5$, $y_{10}$, $y_{15}$). In turn this parity is determined by the parity of the 32 bits ($x_0$, $x_5$, $x_{10}$, $x_{15}$). Overall, the parity of the first column of the AES round output **V** should be compared to the parity of the 32-bit input ($x_0$, $x_5$, $x_{10}$, $x_{15}$) modified by the parities of the four participating s-boxes and the corresponding 32-bits of the round key. Table 2 summarizes the four pairs of 32-bit inputs and the corresponding 32-bit outputs whose parities have to be compared.

Table 2: 32-bit AES round inputs, outputs whose parities should be compared. $x_i$ and $y_i$ are the input and output bytes. Shift row operation shuffles the bytes and hence the input and output bytes do not have identical indices.

| Modify the parity of the 32-bit input | and compare it with actual parity of the 32-bit output |
|---|---|
| ($x_0$, $x_5$, $x_{10}$, $x_{15}$) | ($v_0$, $v_1$, $v_2$, $v_3$) |
| ($x_4$, $x_9$, $x_{14}$, $x_3$) | ($v_4$, $v_5$, $v_6$, $v_7$) |
| ($x_8$, $x_{13}$, $x_2$, $x_7$) | ($v_8$, $v_9$, $v_{10}$, $v_{11}$) |
| ($x_{12}$, $x_1$, $x_6$, $x_{11}$) | ($v_{12}$, $v_{13}$, $v_{14}$, $v_{15}$) |

From Table 2, we can see that input bytes $x_0$, $x_5$, $x_{10}$, and $x_{15}$ cannot be mapped to the same dual-port ROM. Similarly, bytes $x_4$, $x_9$, $x_{14}$ and $x_3$ cannot be mapped to the same dual-port ROM and so on. In our implementation, the byte pairs ($x_0$, $x_1$), ($x_2$, $x_3$), ($x_4$, $x_5$), ($x_6$, $x_7$), ($x_8$, $x_9$), ($x_{10}$, $x_{11}$), ($x_{12}$, $x_{13}$), ($x_{14}$, $x_{15}$) share a dual-port ROM. Since the sixteen s-boxes are implemented using 8 block RAMs, this optimized FPGA implementation uses only 1606 LUTs and a clock period of 16.649 ns.

## 6. Conclusions

In this paper we presented a new low-cost method for concurrent checking for the AES encryption by using parity checking. This technique is applicable to AES decryption as well. The step-by-step modification of the input parity into output parity yields <8% area overhead since the Shift-Rows and Mix Column steps are parity preserving. Also, a single parity tree is used to compute the input and the output parities. We showed how this basic technique can be optimized to exploit the dual-port RAMs on a Xilinx Virtex FPGA. We showed that in such a dual-port RAM-based implementation, the sixteen AES s-boxes should be carefully mapped so as not to compromise the error detection capability. The method can detect technical faults during normal operation and deliberately injected faults. On detecting a fault(s), the stored key may be refreshed to prevent the attacker from uncovering the key.

## 7. References

[1] J. Daemen and V. Rijmen, "AES proposal: Rijndael", csrc.nist.gov/CryptoToolkit/aes/rijndael/

[2] D. Boneh, R. DeMillo and R. Lipton, "On the importance of checking cryptographic protocols for faults", *Proceedings of Eurocrypt*, Lecture Notes in Computer Science vol 1233, Springer-Verlag, pp. 37-51, 1997.

[3] E. Biham and A. Shamir, "Differential Fault Analysis of Secret Key Cryptosystems", *Proceedings of Crypto*, Aug 1997.

[4] J. Bloemer and J.-P. Seifert, "Fault based cryptanalysis of the Advanced Encryption Standard," www.iacr.org/ eprint/2002/075.pdf.

[5] C. Giraud, "Differential Fault Analysis on AES", eprint.iacr.org /2003/008.ps

[6] R Karri, K. Wu, P. Mishra and Y. Kim, "Concurrent Error Detection of Fault Based Side-Channel Cryptanalysis of 128-Bit Symmetric Block Ciphers," *IEEE Transactions on CAD*, Dec 2002.

[7] R. Karri, G. Kuznetsov and M. Goessel, "Parity-based Concurrent Error Detection of Substitution-Permutation Network Block Ciphers," *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems*, Springer Verlag LNCS 2779, Sep 2003.

[8] R. Karri, M. Goessel, and G. Kousnezow, "Method for error detection in kryptographic substitution permutation networks," patent application pending.

[9] G. Bertoni, L. Breveglieri, I. Koren and V. Piuri, "On the propagation of faults and their detection in a hardware implementation of the advanced encryption standard," *Proceedings of ASAP'02*, pp. 303-312, 2002.

[10] G. Bertoni, L. Breveglieri, I. Koren, and V. Piuri, "Error Analysis and Detection Procedures for a Hardware Implementation of the Advanced Encryption Standard," *IEEE Transactions on Computers*, vol. 52, No. 4, pp. 492-505, Apr 2003.

[11] H. Heys and S. E. Tavares, "Avalanche characteristics of substitution permutation encryption networks," *IEEE Transactions on Computers*, vol. 44, no. 9, pp. 1131-1139, Sep 1995.

[12] Vl. Moshanin, V. Otscheretnij and A. Dmitriev, "The Impact of Logic Optimization on Concurrent Error Detection," *Proceedings of the 4th IEEE International On-Line Testing Workshop*, pp. 81-84, Jul 1998.

[13] P. Chodowiec and K. Gaj, "Very Compact FPGA Implementation of the AES Algorithm," *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems*, Springer Verlag LNCS 2779, pp. 319-333, Sep 2003

[14] F. Standaert, G. Rouvroy, J. Quisquater and J. Legat, "Efficient Implementation of Rijndael Encryption in Reconfigurable Hardware: Improvements and Design Trade-offs," *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems*, Springer Verlag LNCS 2779, pp. 343-350, Sep 2003.