9-18-2014 12:00 AM

A Multiple Bit Parity Fault Detection Scheme for The Advanced Encryption Standard Galois/Counter Mode

Amir Hossein Ali Kouzeh Geran, *The University of Western Ontario*

Supervisor: Dr. Arash Reyhani Masoleh, *The University of Western Ontario*

A thesis submitted in partial fulfillment of the requirements for the Master of Engineering Science degree in Electrical and Computer Engineering

A MULTIPLE BIT PARITY FAULT DETECTION SCHEME FOR THE ADVANCED ENCRYPTION STANDARD GALOIS/COUNTER MODE

by

Amir Hossein Ali Kouzeh Geran

Graduate Program in Electrical and Computer Engineering

A thesis submitted in partial fulfillment
of the requirements for the degree of
Masters of Science

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

Abstract

The Advanced Encryption Standard (AES) is a symmetric-key block cipher for electronic data announced by the U.S. National Institute of Standards and Technology ($NIST$) in 2001. The encryption process is based on symmetric key (using the same key for both encryption and decryption) for block encryption of 128, 192, and 256 bits in size. AES and its standardized authentication Galois/Counter Mode (GCM) have been adopted in numerous security-based applications. GCM is a mode of operation for AES symmetric key cryptographic block ciphers, which has been selected for its high throughput rates in high speed communication channels.

The GCM is an algorithm for authenticated encryption to provide both data authenticity and confidentiality that can be achieved with reasonable hardware resources. The hardware implementation of the AES-GCM demands tremendous amount of logic blocks and gates. Due to natural faults or intrusion attacks, faulty outputs in different logic blocks of the AES-GCM module results in erroneous output. There exist plenty of specific literature on methods of fault detection in the AES section of the AES-GCM.

In this thesis, we consider a novel fault detection of the GCM section using parity prediction. For the purpose of fault detection in GCM, two independent methods are proposed. First, a new technique of fault detection using parity prediction for the entire GCM loop is presented. Then, matrix based CRC multiple-bit parity prediction schemes are developed and implemented. As a

result, we achieve the fault coverage of about 99% with the longest path delay and area overhead of 23% and 10.9% respectively. The false alarm is 0.12% which can be ignored based on the number of injected faults.

Keywords: Fault Detection, Parity Prediction, AES-GCM, Matrix Based CRC

Acknowledements

I would like to express my sincere appreciation to Prof. Arash Reyhani-Masoleh for his extra care, great supervision and guidance during my studies and preparation of this thesis.

Amir Ali Kouzeh Geran

London, ON

2014

Contents

List of Figures

List of Tables

List of Appendices

Chapter 1

Thesis Contributions and Outline

1.1 Introduction

1.1.1 Advanced Encryption Standard(AES)

The Advanced Encryption Standard (AES) has been widely used in cryptosystems after it was introduced by *NIST* [14] in 2001. It was a replacement for Data Encryption Standard (DES) [6]. Since then, many different hardware and software projects have been developed and implemented to obtain more efficient area and timing results.

1.1.2 Galois/counter Mode(GCM)

The Galois Counter Mode of operation (GCM) [1] is a combined encryption and authentication process introduced by David Mcgrew and John Viega [22]. The mode is defined in NIST SP 800-38D [35]. The GCM is a mode of operation that uses a universal hash function over a binary Galois field to provide assurance of the authenticity and the confidentiality of data in its encryption/decryption. The GCM can also provide authentication assurance for addi-

tional data that is not encrypted. In particular, GCM can detect both accidental modifications of the data, or unauthorized alterations to ensure proper authentication while protects confidentiality to make the data readable by intended receiver. It could be implemented into the hardware to achieve high speed operation with low cost and low latency.

The AES-GCM has been widely used in networking communications. In general, fault detection techniques can be very useful towards the protection of encryption and malicious attack prevention. The GCM is designed to support very high data rates due to pipelining and parallel processing techniques as well as high degree of authenticity and confidentiality. This will result in authenticated encryption at data rates of many tens of $Gbps$, permitting high grade encryption and authentication on communication systems. Recently, the GCM is being used in lower data rate applications. Therefore, much more reliable fault detection techniques are needed in industry level.

1.2 Motivation and Scope of Thesis

There are two sources of faults in cryptography systems, natural faults and fault attacks. The natural faults are caused by physical defects in the ASIC or the electrical circuit malfunction. Four common types of defects occur in logic gates during the fabrication or due to physical failure i.e., the bridging or short circuit between adjacent lines, breaks or open circuits, or permanently adopting

logic 0 or logic 1 which is modeled by stuck-at-0 and stuck-at-1 respectively (in general stuck-at-fault). The stuck-at-fault model assumes that only one input or output on each gate will be faulty at a time. Assuming that if more than one are faulty, a test that can detect any single fault, should easily find multiple faults.

The intruder attack consists of a series of fault injection into the system to obtain any leakage of secret information, this type of transient fault should also be detected to prevent such attacks. Therefore, the need for a robust fault detection method is highly demanded to have much more protection for the integrity and authenticity of data over the communication channels. As soon as the attacker inject the fault into the system, the fault detection module generates an error signal to prevent the system to proceed to the next level. Thus, the attacker won't be able to complete the attack sequence. This thesis aims to create a reliable GCM module which is capable of detecting permanent and transient faults. To make a system more reliable against faults, there have been three different approaches towards the fault detection [5]. These methods are hardware, time, and information redundancy. In hardware redundancy technique, one can duplicate hardware to the system for fault detection which causes 100% fault detection versus 100% area overhead. In time redundancy approach, the function of hardware is evaluated in different time slices to detect transient faults. Again, one can obtain 100 percent fault coverage versus 100% delay increase. The information redundancy deals with additional extra added information like

parity bits to the system to locate the faults.

The goal of this thesis is to introduce a novel method of fault detection in the GCM module using the information redundancy technique by adding parity bits to the Circuit Under Test (CUT).

1.3 Contributions

In this thesis, we have introduced a novel matrix based CRC fault detection architecture using multiple bit parity prediction method for entire the GCM loop with high rate of fault coverage which is about 99%. The proposed scheme is generic and the number of parity bits can be adjusted based on the available resources and needed fault coverage. The proposed fault detection schemes can be applied as an universal method of fault detection because of its unique processing of CRC pattern generation. The proposed fault injection using VHDL and Tcl programming makes the design verification and testing easier, faster, and more reliable. The contributions of the thesis are summarized as follows:

- A new fault detection scheme for the entire GCM module using the GCM charactristics defined by NIST [35] and Galois field principles including the formulations and block diagram.

- New approach in using CRC method to generate fault detection patterns based on the number of the used parity bits. All formulation, CRC patterns and implementation are covered.

- A new method of fault injection in the GCM module using the VHDL is proposed. Both the GCM and fault injection are implemented in VHDL.
- Tcl stimulus script is used to activate the fault injection for simulation of proposed fault detection to investigate the fault coverage.
- Implementation of the fault detection scheme is performed on FPGA for area overhead and timing analysis. This thesis outlines an accurate and reliable GCM module which detects all types of faults in each clock cycle to prevent sending false information through the communication channels.

1.4 Thesis Outline

We have developed a formal model in hardware that allows us to formulate the fault detection problem for arbitrary permanent and transient faults in the entire GCM loop. In Chapter 2, we explain the Authenticated encryption/decryption in GCM, inputs and outputs of GCM, and the GCM block diagram. We also outline the principles of Galois field and previous work has been done on AES and multipliers fault detection. In Chapter 3, we have introduced a novel parity prediction scheme for entire GCM module using the properties of GCM and application of Galois field principles. In Chapter 4, we have established a reliable matrix based CRC (Cyclic Redundancy Check) multi- bit parity prediction. The proposed scheme is generic in terms of the number of used parity bits. If we increase the number of parity bits, we can achieve close to 100% accuracy

in detecting permanent and transient faults. Chapter 5 depicts the simulation results and implementation in FPGA as well as timing and overhead analysis. A new method of fault injection in the GCM module using VHDL language and stimulus script programming called Tcl are given in Appendix A&B.

Chapter 2

Priliminaries and Literature Review

In this chapter, we have an overview on the Galois field principles, GCM characteristics and operation, AES operation and related fault detection, and the multiplier module used in GCM block. The bit-parallel multiplier and the fault detection in the multiplier module will be discussed. The multiplier consists of tremendous amount of gates which could generate the faulty output due to natural or transient faults in any part of its structure. The area overhead of the multiplier in the GCM is up to 30% of the total space [38]. Therefore, the selection of low complexity multiplier contributes towards the final cost and the operating frequency.

2.1 Finite Field

Finite field arithmetic [20], [19] has become prominent solution in different applications like cryptography and digital communication systems. In the GCM implementation, each ciphertext is treated as an element of a finite field. During the Tag generation process, each ciphertext or element of the field is multiplied

by *Hash Key H*, or added to another field element. The element H is considered constant field element which does not change until the next encryption.

2.1.1 Principles of Galois Field

Only binary field is used in the GCM. Therefore, we explain the Galois Field $GF(2^m)$ which is extensively employed in the GCM implementation. Thus, we mainly focus on binary extension fields in thesis. The number of elements in the field is equal to 2^m. The elements of the field are represented by polynomials with coefficients belonging to $GF(2)$. The elements of the field are generated by selecting irreducible polynomial $F(x)$ which cannot be factored into any polynomials in $GF(2^m)$ [23]. All the elements in $GF(2^m)$ are represented by polynomials modulo $F(x)$. The multiplication is done modulo the selected irreducible polynomial.

2.1.2 Finite Field Arithmetic

Addition In Galois Field

Let A and B represent two elements in $GF(2^m)$. Then, one can write each element according to its polynomial representation:

$$\begin{aligned} A &= \sum_{i=0}^{m-1} a_i x^i = a_{m-1}x^{m-1} + \cdots + a_0 \quad a_i \in \{0, 1\} \\ B &= \sum_{i=0}^{m-1} b_i x^i = b_{m-1}x^{m-1} + \cdots + b_0 \quad b_i \in \{0, 1\} \end{aligned} \tag{2.1}$$

The addition is componentwise sum of each element in $GF(2^m)$ over $GF(2)$ which can be written as follows

$$A + B = \sum_{i=0}^{m-1} (a_i + b_i)x^i \tag{2.2}$$

Therefore, addition is performed by bitwise exclusive-or (XOR) of the elements of the field. Thus, we represent all addition with + sign hereafter which denotes XOR operation on coordinates of field elements.

Multiplication In Galois Field

The result of multiplication of A and B defined in (2.1) and denoted by S is a polynomial with order of $2m - 2$ shown as

$$S = AB = s_{2m-2}x^{2m-2} + \cdots + s_1x^1 + s_0 \tag{2.3}$$

It is reduced to the order of $m - 1$ after modular reduction, if x is the root of irreducible polynomial generating the field elements, then the final result of multiplication depends solely on the chosen irreducible polynomial.

$$\begin{aligned} C &= AB \ mod \ F(x) \ \in GF(2^m) \\ C &= c_{m-1}x^{m-1} + \cdots + c_0 \end{aligned} \tag{2.4}$$

For the purpose of this thesis, we use the recommended irreducible polynomial [35] for the GCM in $GF(2^{128})$ by the following pentonomial

$$f(x) = x^{128} + x^7 + x^2 + x + 1 \tag{2.5}$$

2.1.3 Example $GF(2^3)$

In order to clarify the concept of irreducible polynomial and the polynomial basis, an example is given in $GF(2^3)$ for simplicity. One can extend the concept to $GF(2^{128})$ which is discussed in the following Chapters.

To define the field structure and elements, we need to define the monic irreducible polynomial $f(x)$ in $GF(2^3)$ over $GF(2)$. The irreducible polynomial $f(x)$ is shown in [19] as

$$f(x) = x^3 + \sum_{i=0}^{2} f_i x^i \qquad f_i \in \{0, 1\} \tag{2.6}$$

It is clear that f_0 must be 1, otherwise $f(x)$ is not irreducible and could be divided by x. The sum of coefficients must be 1, otherwise $x+1$ will be a factor and $f(x)$ becomes reducible. Thus, $1 + f_2 + f_1 + 1$ must be 1. Therefore, f_1 or f_2 must be 0. As result, we outline the following monic polynomials in $GF(2^3)$.

$$x^3 + x^2 + 1 \quad \text{and} \quad x^3 + x + 1 \tag{2.7}$$

Either polynomials can be used to define the the elements of the field. If α is the root of the polynomial $f(x)$, then $f(\alpha) = 0$ and $\alpha^3 = \alpha^2 + 1$. Now, we

calculate $\alpha^i s$ shown in Table 2.1.

i	α^i	*Binary*	$\alpha^i \, mod \, f(\alpha)$
0	1	001	1
1	2	010	α
2	4	100	α^2
3	5	101	$\alpha^2 + 1$
4	7	111	$\alpha^2 + \alpha + 1$
5	3	011	$\alpha + 1$
6	6	110	$\alpha^2 + \alpha$
7	1	001	1

Table 2.1: $\alpha^i s$ in $GF(8)$ based on $x^3 + x^2 + 1$.

We can easily show that every nonzero element of $GF(2^3)$ is a power of α and also the linear combination of the polynomial basis $\{1, \alpha, \alpha^2\}$. Here, α is called primitive element. The selection of irreducible polynomial leads to the maximum number of distinguished elements in the related field. Therefore, one can process eight different field elements out of 3 bit length in the above example of $GF(2^3)$.

As an example multiplication of two field elements 010 and 100 is α^3 which results another field element 101.

2.2 The GCM Operation

The GCM performs authentication and encryption of data at high speeds for both software and hardware implementations. Data integrity is achieved by Galois Field (GF) multiplication operations while a symmetric key block ci-

pher named Advanced Encryption Standard (AES) is used for the purpose of confidentiality. The GCM uses block cipher for authenticity. A block cipher is a symmetric key cipher operating on fixed-length groups of bits, called blocks [27]. The mode of operation defines how to repeatedly apply a cipher's block operation to transform amounts of whole plaintext which has been divided into fix-sized blocks. The GCM requires a unique IV for each encryption operation. The IV must be random and should not repeat for other encryption. This leads to generation of different ciphertexts even the same plaintext is encrypted multiple times with the same key. The security of AES-GCM depends on the freshness of the nonce/key combination. Thus, we cannot use statically configured keys. Instead, an automated key management system is implemented. Authors in [4] have discussed four general key management techniques i.e., Key Transport, Key Agreement, Key-Encryption Keys, and Passwords. The GCM could be implemented in pipeline form, which results in throughput of more than $10\,Gbps$. While in Tag generation process of the GCM, a chained Galois multiplication is used, the sequential data ($Ciphertext$) can be fed in through the pipeline form.

2.2.1 Encryption/Decryption in GCM

There are four inputs for the authenticated encryption, a secret key K with the length based on the block cipher. An initialization vector IV, that can be any

number between 1 and 2^{64} that is unique for each application. A plaintext P, that can have any number of bits between 0 and $2^{39} - 256$. Additional authenticated data (AAD) denoted by A which is authenticated by the GCM module, but not encrypted. AAD includes version numbers, port, address or other field information can be a number between 0 and 2^{64}. There are two outputs: A ciphertext C with the same length as the plain text P. An authentication tag T, whose length can be any value between 0 and 128.

The authenticated decryption operation has five inputs: K, IV, C, A, and T. It either generates the plaintext P, or a FAIL signal once the inputs are not authentic for the shared key.

Encryption Process

The plaintext P is divided into the blocks of 128 bit long. Let n and u be two positive integers, then the total number of bits in P can be written as $(n-1)128+u$ where $1 \leq u \leq 128$. As a result, P can be shown as $P_1, P_2, \ldots, P_{n-1}, P_n^*$, which are called data blocks with the length of 128 bits except for the last block that could be less than 128 bits.

Similarly, the ciphertext C is represented as $C_1, C_2, \ldots, C_{n-1}, C_n^*$. The additional authenticated data A is also represented as $A_1, A_2, \ldots, A_{m-1}, A_m^*$. The total number of bits required for A can be written as $(m-1)128+v$, m and v are two positive integers where $1 \leq v \leq 128$.

The inputs A and C are formatted as above and the function $GHASH$ is

defined by $X_{m+n+1} = GHAS\,H(H, A, C)$ as

$$X_i = \begin{cases} 0 & for\, i = 0 \\ (X_{i-1} \oplus A_i) \cdot H & for\, i = 1, \ldots, m-1 \\ (X_{m-1} \oplus (A_m^{\star} \parallel 0^{128-v})) \cdot H & for\, i = m \\ (X_{i-1} \oplus C_{i-m}) \cdot H & for\, i = m+1, \ldots, m+n-1 \\ (X_{m+n-1} \oplus (C_n^{\star} \parallel 0^{128-u})) \cdot H & for\, i = m+n \\ (X_{m+n} \oplus (len(A) \| len(C)) \cdot H & for\, i = m+n+1 \end{cases} \tag{2.8}$$

Two operations are used in (2.8) denoted by " $\oplus$ " for XOR and " $\cdot$ " for the finite field multiplication over $GF(2^{128})$ which is explained in previous section. The *len*() function is used for generating the length block, its function is to compute the total number of bits in the operand and returns a 64 bit value. AAD and the ciphertext are applied to the *len* function, then the result is concatenated to create the length block.

The sets of equation to define the authenticated encryption operation are defined in [35] as follows

$$\begin{aligned} H &= E(K, 0^{128}) \\ U_0 &= \begin{cases} IV \parallel 0^{31}1 & if\ \ len(IV) = 96 \\ GHAS\,H(H, \{\}, IV) & otherwise \end{cases} \\ U_i &= incr(U_{i-1}) \qquad for\, i = 1, \ldots, n \\ C_i &= P_i \oplus E(K, U_i) \qquad for\, i = 1, \ldots, n \\ T &= MS\,B_t(GHAS\,H(H, A, C) \oplus E(K, U_0) \end{aligned} \tag{2.9}$$

In (2.9), "$Hash\,Key$" H is generated by the AES encryption of a 128 bit

block of all zero. the *incr*() function increments the value of the the counter U by one in each clock cycle. This value will be *XORed* with $E(K, U_0)$ which is the AES encryption of shared key (K) and the initial counter value U_0. At the final stage, the Authentication Tag T is generated by choosing the t most significant bits of the result. The initial value of the counter is $U_0 = (IV \parallel 0^{31}1)$ if $len(IV) = 96$, otherwise $GHASH(H, \{\}, IV)$. The counter will be incremented by one at each clock cycle using the *incr*() function. Then, the AES encrypted value of the counter will be added to the plaintext P_i to generate the ciphertext C_i which will be applied to the $GHASH$ calculator in (2.8).

2.2.2 The GCM Block Diagram

As stated in previous section, the main part of the $GHASH$ function in the GCM is multiplication of $(X_{i-1} \oplus C_i)$ by H in $GF(2^{128})$. Figure 2.1 illustrates the block diagram for GCM. In this scheme, we consider sequential structure for the multiplier and apply A_i or C_i and H in block-length of 128 in serial to $GHASH$ calculator in (2.8). In this figure, the 128-bit register $Y = (y_{127}, ..., y_2, y_1, y_0)$ will be initialized by all zeros at the beginning of the clock cycle. Let $Y^{(n)}$ denote the contents of Y at the *nth* clock cycle. Let X_i be the multiplier output at *ith* clock cycle. Furthermore, the initial value of register $X_0 = Y^{(0)} = (0, \ldots, 0)$. Thus, the content of Y after the first clock cycle shows as $X_1 = Y^{(1)} = C_1 \cdot H$, in the second clock cycle we obtain $X_2 = Y^{(2)} = (C_1 \cdot H + C_2) \cdot H$, and the loop continues until we extract X_{m+n+1}

in the $(m + n + 1)th$ clock cycle as defined in (2.8).

A_i or encrypted P_i can be selected through the multiplexer based on the values of m and n .

2.2.3 Review on GHASH

Software Implementation of High Performance GHASH Algorithms

Authors in [34] provide an efficient way of software implementation of high performance GHASH function and also on the implementation of GHASH using a carry-less multiplication instruction supplied by Intel. The work includes implementation of the high performance GHASH and its comparison to the standard implementation of GHASH function. It also includes comparison of the two implementations using Intels carry-less multiplication instruction. The proposed software implementations suggest that the new GHASH algorithm can't take advantage of the Intel carry-less multiplication instruction PCLMULQDQ. The work shows that the implementations done without using the PCLMULQDQ instruction performs better. This suggest that the new algorithm will perform better on embedded systems that do not support PCLMULQDQ.

High Performance GHASH Function for Long Messages

Authors in [27] present a new method to compute the GHASH function. AS the GHASH calculations consist of n successive multiply and addition over

$GF(2^{128})$ for a bit string made of n blocks of 128 bits each. In this work, they propose a method to replace all but a fixed number of those multiplications by additions on the field. This is achieved by using the characteristic polynomial of H. They present how to use this polynomial to speed up the GHASH function and how to efficiently compute it for each session that uses a new H.

An Architecture for the AES-GCM Security Standard

Authors in [37] present a fully pipelined and parallelized hardware architecture for AES-GCM. The results from this thesis show that the round transformations of confidentiality and hash operations of authentication in AES-GCM can cooperate very efficiently within this pipelined architecture. Furthermore, this AES-GCM hardware architecture never unnecessarily stalls data pipelines. This thesis provides a complete FPGA-based high speed architecture for the AES-GCM standard, suitable for high speed embedded applications.

2.3 The AES Operation

AES [33], [13] is a symmetric block cipher with block-length of 128 bits. The size of *Key* can be chosen from 128 bits, 192 bits, or 256 bits. Thus, the AES-128 uses 10 rounds of operations, the AES-192 with 12 rounds of operations, and AES-256 with 14 rounds of operations. The AES divides the plaintext into 16 bytes (128 bits). Thus, each block of 128 bits form an State array of 4×4. The AES round functions perform the collection of GF operations:

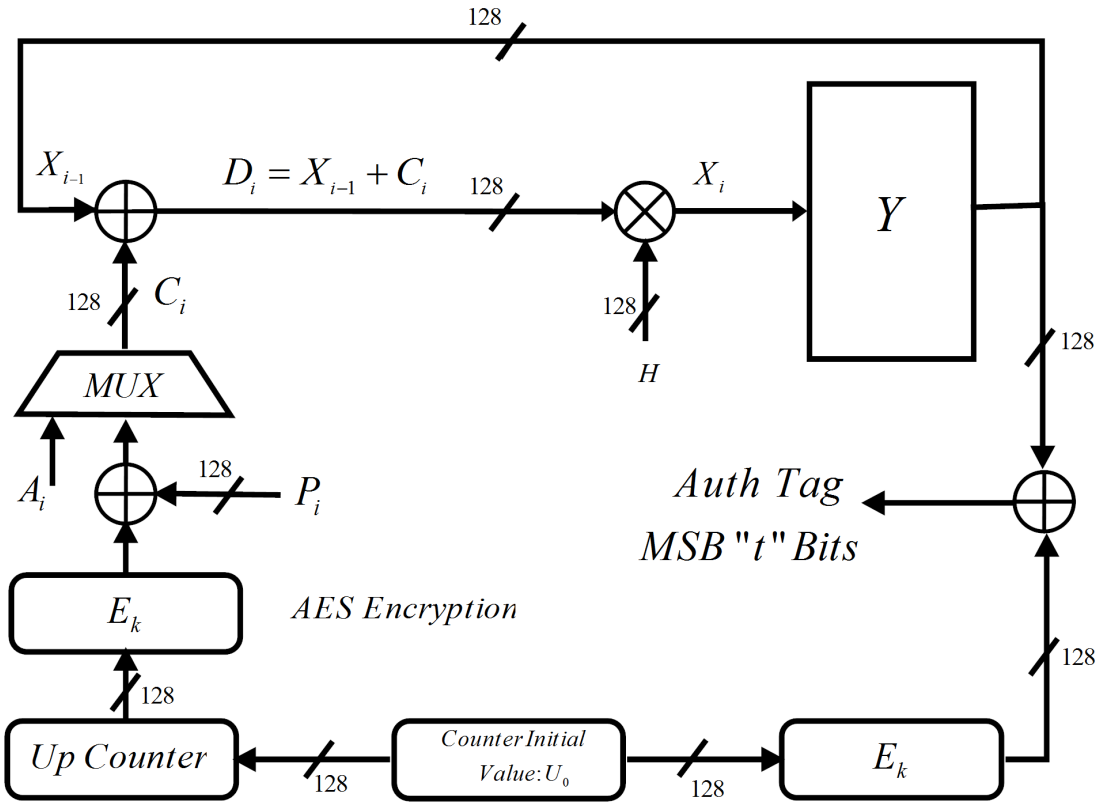

Figure 2.1: Block Diagram of AES-GCM

- SubBytes(*GF* inverse): The processing is done on each byte through an S-Box which is a substitution table, where one byte is replaced with another byte, based on a substitution algorithm. The SubBytes process scrambles each byte.

- ShiftRows: The process is mixing data within rows. Row zero of the State is not shifted, row 1 is shifted 1 byte, row 2 is shifted 2 bytes, and row 3 is shifted 3 bytes. The ShiftRows process scrambles each row.

- MixColumns(*GF* matrix multiplication): The process is mixing data within columns. The 4 bytes of each column in the State are exchanged with another 4-byte number through the finite field arithmetic. The MixColumns process scrambles each column.

- AddRoundKey(*GF* addition): The encryption process is performed in this part, when each byte in the current State is *XOR*ed with the subkey. The subkey is formed based on specification of FIPS [13].

Therefore, one round of AES on 128 bit plaintext consists of performing SubBytes, ShiftRows, MixColumns, AddRoundKey. Then, the ciphertext with the length of 128 bit is obtained.

Decryption is done through inverse AES functions. Thus, we perform the followings for decryption: AddRoundKey, inverseMixColumn, inverseShiftRows, inverseSubByte.

2.3.1 Fault Attacks and Detection in AES

Fault Attacks

As an example the authors in [15] show the fault attacks in the form of transient fault on symmetric cryptosystems like AES have the following outcome:

- Modification of 1 byte of the Mix Columns input has an impact on 4 bytes.
- Modification of 1 byte between the Mix Columns of the 7th round and the Mix Columns of the 8th round.
- The secret key can be recovered by using 2 faulty ciphertexts.

Therefore, the need for a robust fault detection method is highly demanded to have much more protection for the integrity and authenticity of data over the communication channels. As soon as the attacker can inject the fault into the system, the fault detection module generates an error signal to prevent the system to proceed to the next level. Thus, the attacker won't be able to complete the attack sequence. This thesis aims to create a reliable GCM module which is capable of detecting permanent and transient faults.

Fault Detection

There exists a large number of papers which offer different methods of fault detection in the AES [24], [29], [39], [5]. The authors in [26] proposed a lightweight concurrent parity-based fault detection scheme for the S-Box using normal basis. This scheme can also be applied to the inverse S-Box. They introduced the least area and delay overhead S-Box and its fault detection scheme for the optimum composite field.

In this regard, high error coverage was achieved. The S-Box is a nonlinear operation which takes an 8-bit input and generates an 8-bit output. In the S-Box, the irreducible polynomial of $f(x) = x^8 + x^4 + x^3 + x + 1$ is used to construct the binary field $GF(2^8)$. Let $X \in GF(2^8)$ and $Y \in GF(2^8)$ be the input and the output of the S-box respectively. Then, the S-Box consists of the multiplicative inversion, i.e., $X^{-1} \in GF(2^8)$, followed by an affine transformation. The affine transformation consists of the matrix A and the vector b to generate the output as $y = Ax^{-1} + b$ where, y and x^{-1} are vectors corresponding to the field elements Y and X^{-1} respectively.

In another section of the paper, the explanation of the composite field realization of the multiplicative inversion using normal basis is discussed.

In the following sections of the referenced paper, the parity-based fault detection scheme of the S-Box using this realization is investigated. For the purpose of fault coverage, the authors use multiple stuck-at fault model at the logic level. This type of fault, which forces multiple nodes to be stuck at logic one

(for stuck-at one) or zero (for stuck-at zero) independent of the fault-free logic values, has been frequently used in the literature. It is noted that the presented scheme is independent of the life time of the faults. Thus, both permanent and transient stuck-at faults lead to the same fault coverage.

In the parity-based fault detection scheme of a block of logic gates, the parity of the block is predicted and it is compared to the actual parity. The result of this comparison is the error indication flag of the corresponding block. This method is utilized in the literature to develop a fault detection scheme for different applications. The authors have divided the S-box into 5 blocks. This results in low overhead parity predictions while maintaining the fault detection required for the security-constrained environments.

Another fault detection scheme has been discussed in [16]. In SubBytes, using the technique of parity to make the prediction parity for the SubBytes is complex due to nonlinearity of this transformation. To protect the SubBytes, the authors use the hardware redundancy method. They implement two SubBytes transformations in parallel. At the end of the SubBytes computation, the results are compared and every discrepancy is considered as a fault.

The same method can be applied in the decryption process by using two InvSubBytes transformation in parallel. In ShiftRows, the output of the SubBytes transformation acts as the input to ShiftRows. Therefore, the output state of ShiftRows is obtained by shifting the matrix state. To secure the ShiftRows transformation, They used the scrambling method. This method consists of

scrambling the output of ShiftRows. In MixColums and AddRoundKey, the authors used cyclic redundancy check (*CRC*) for fault detection. The comparison of *CRC* is made before and after each operation to detect the faults. They showed more than 99% fault coverage with the area overhead of 22.51% and frequency degradation of 13.86%.

2.4 Finite Field Multipliers In AES-GCM

In this section, we investigate the different types of multipliers and related area and time complexity. The choice of the multiplier type in the AES-GCM depends solely on the speed and area constrains of the application. There are three different types of multipliers that can be used in the AES-GCM. These are explained below. It is noted that in this thesis we have used the bit-parallel one.

Bit-parallel Multipliers:

In this multiplier the inputs are being applied in word format with length of m and the output will be the same word length. The modular reduction is applied at the same time to obtain the field element as output. The area complexity in this multiplier is $O(m^2)$ and requires 1 clock cycles to accomplish the result. Block diagram of bit-parallel multiplier is shown in Figure 2.2.a.

Bit-Level Multipliers:

Both multiplicand and multiplier are fed bit by bit through the shift register to the multiplier input. There are two methods of applying the digits, MSB-first and LBS-first. The area complexity is $O(m)$ and requires m clock cycles for completion. Block diagram of bit-level multiplier is shown in Figure 2.2.b.

Digit-Level Multiplier:

In this method, the operands are divided into k digit length to be applied to the multiplier input . The area complexity is $O(km)$ and required time for completion is $\frac{m}{k}$ clock cycles. Block diagram of digit-level multiplier is shown in Figure 2.2.c.

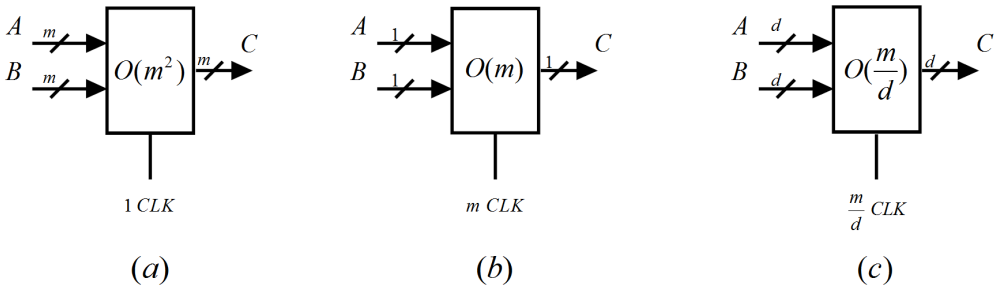

Figure 2.2: Block Diagram of Multipliers a. Parallel b. Bit-level c. Digit-level

2.4.1 Low Complexity Multiplier in $GF(2^m)$

In this section, we have an overview on a low complexity bit-parallel multiplier. As mentioned, the multiplier plays an important role in AES-GCM module.

Therefore, the type of the chosen multiplier contributes towards the area and operating frequency of the said module. The authors in [31] have shown an approach towards low complexity bit parallel multiplier which will be used in the following sections for the purpose of fault analysis.

They have shown that $C = AB \in GF(2^{128})$ can be written as

$$\mathbf{c} = (\mathbf{L} + \mathbf{Q}^T\mathbf{U})\mathbf{b} \tag{2.10}$$

Where $\mathbf{b} = [b_0, b_1, \cdots, b_{m-1}]^T$ and T denotes transposition of vector. $\mathbf{L}$ and $\mathbf{U}$ are two Toeplitz matrices whose elements consist of $a_i s$ which are the coordinates of A. $\mathbf{L}$ is $m \times m$ lower triangular matrix and $\mathbf{U}$ is $(m-1) \times m$ upper triangular matrix shown in Figure 2.3.

$$\mathbf{L} = \begin{bmatrix} a_0 & 0 & \dots & 0 & 0 & 0 \\ a_1 & a_0 & \cdots & 0 & 0 & 0 \\ a_2 & a_1 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ a_{m-2} & a_{m-3} & \dots & a_1 & a_0 & 0 \\ a_{m-1} & a_{m-2} & \dots & a_2 & a_1 & a_0 \end{bmatrix}, \mathbf{U} = \begin{bmatrix} 0 & a_{m-1} & a_{m-2} & \dots & a_2 & a_1 \\ 0 & 0 & a_{m-1} & \dots & a_3 & a_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & a_{m-1} & a_{m-2} \\ 0 & 0 & 0 & \dots & 0 & a_{m-1} \end{bmatrix}, \mathbf{b} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{m-2} \\ b_{m-1} \end{bmatrix}$$

Figure 2.3: Demonstration of **L**, **U**, and **b** in Matrix Presentation.

Q is called the reduction matrix which is $(m-1)\times m$ and could be found through the following relationship

$$\alpha \uparrow = \mathbf{Q}\alpha \ mod \ F(\alpha), \tag{2.11}$$

where $\alpha \uparrow = [\alpha^m, \alpha^{m+1}, \cdots, \alpha^{2m-2}]^T$ and $\alpha = [1, \alpha, \alpha^2, \cdots, \alpha^{m-1}]^T$.

Using (2.11), one can obtain the reduction matrix **Q** for the GCM in order to figure out the single and multi-bit parity-based fault detection scheme. Assuming α is the root of irreducible polynomial, then $f(\alpha) = 0$ and we could rewrite (2.5) as follows

$$\alpha^{128} + \alpha^7 + \alpha^2 + \alpha + 1 = 0. \tag{2.12}$$

As in the finite field, $-1 = +1$ then one can derive the following equation

$$\alpha^{128} = 1 + \alpha + \alpha^2 + \alpha^7. \tag{2.13}$$

Then, the first row of the reduction matrix **Q** is obtained as follows

$$\alpha^{128} = \underbrace{\begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & \cdots & 0 \end{bmatrix}}_{1\times 128} \begin{bmatrix} 1 \\ \alpha \\ \alpha^2 \\ \vdots \\ \alpha^{127} \end{bmatrix} \tag{2.14}$$

Next, we calculate the consecutive powers of α as

$$\begin{aligned}
\alpha^{i+128} &= \alpha^i + \alpha^{i+1} + \alpha^{i+3} + \alpha^{i+8} \qquad for\ 1 \leq i \leq 120 \\
\alpha^{129} &= \alpha + \alpha^2 + \alpha^3 + \alpha^8 \\
&\vdots \\
\alpha^{248} &= \alpha^{120} + \alpha^{121} + \alpha^{122} + \alpha^{127}
\end{aligned} \tag{2.15}$$

We can outline α^{249} as follows

$$\alpha^{249} = \alpha^{121} + \alpha^{122} + \alpha^{123} + \alpha^{128}. \tag{2.16}$$

Substituting α^{128} from (2.16), one can find

$$\begin{aligned}
\alpha^{249} &= 1 + \alpha + \alpha^2 + \alpha^7 + \alpha^{121} + ^{122} + \alpha^{123} \\
&\vdots \\
\alpha^{253} &= \alpha^4 + \alpha^5 + \alpha^6 + \alpha^{11} + \alpha^{125} + \alpha^{126} + \alpha^{127}
\end{aligned} \tag{2.17}$$

As the final term is $2m - 2 = 254$, then for α^{254} by using (2.13) we obtain

$$\alpha^{254} = 1 + \alpha + \alpha^2 + \alpha^5 + \alpha^6 + \alpha^{12} + \alpha^{126} + \alpha^{127} \tag{2.18}$$

Therefore, the $\mathbf{Q}$ matrix could be written using (2.13) as shown in Figure 2.4, where $R1\ to\ R127$ and $C1\ to\ C128$ denote the row and column numbers respectively.

From completed $\mathbf{Q}$ matrix, the $\mathbf{Q}^T$could be written as shown in Figure 2.5. Now, we rewrite (2.10) in matrix presentation as

$$\begin{array}{c}
\\
R1\\
R2\\
R3\\
\vdots\\
R121\\
R122\\
\vdots\\
R126\\
R127
\end{array}
\begin{bmatrix}
C1 & & & & & & & & & & & & C13 & & C121 & & & & & & & C128\\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
\vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1\\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0\\
\vdots & \ddots & \ddots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & 0 & 0 & 0 & \cdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots & \vdots\\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1\\
1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1
\end{bmatrix}$$

Figure 2.4: Demonstration of **Q** in matrix presentation

$$\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{bmatrix} = (\mathbf{L} + \mathbf{Q}^T\mathbf{U}) \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{m-1} \end{bmatrix} \tag{2.19}$$

Hereafter, we name matrix $\mathbf{E}$ as $\mathbf{E} = \mathbf{L} + \mathbf{Q}^T\mathbf{U}$. As in the GCM module, one of the multiplicands is always power of the *Hash Key* H, then $\mathbf{L} = z(H)$ and $\mathbf{U} = g(H)$; Therefore, $\mathbf{E}$ is a function of H e.g. $\mathbf{F} = w(H)$. In the equation $\mathbf{c} = \mathbf{Eb}$, we realize that the output of multiplier will be the function of H and $\mathbf{b}$. Thus, we use this characteristic to outline parity prediction module in terms of the GCM elements and then define the single or multi-bit parity prediction scheme in the entire GCM module in Chapter 4.

2.5 Fault Detection In Multiplier

There exists a large number of articles on fault detection in finite field multipli-

	C1			C4		C121					C127
R1	1	0	0	0	⋯	0	1	0	0	0	1
R2	1	1	0	0	⋯	0	1	1	0	0	1
R3	1	1	1	0	⋯	0	1	1	1	0	1
R4	0	1	1	1	⋯	0	0	1	1	1	0
R5	0	0	1	1	⋯	0	0	0	1	1	0
R6	0	0	0	1	⋯	0	0	0	0	1	1
R7	0	0	0	0	⋯	0	0	0	0	0	1
R8	1	0	0	0	⋯	0	1	0	0	0	0
R9	0	1	0	0	⋯	0	0	1	0	0	0
R10	0	0	1	0	⋯	0	0	0	1	0	0
R11	0	0	0	1	⋯	0	0	0	0	1	0
R12	0	0	0	0	⋯	0	0	0	0	0	1
R13	0	0	0	0	⋯	0	0	0	0	0	0
⋮	⋮	⋮	⋮	⋮	⋯	⋮	⋮	⋮	⋮	⋮	⋮
R121	0	0	0	0	⋯	1	0	0	0	0	0
R122	0	0	0	0	⋯	1	1	0	0	0	0
R123	0	0	0	0	⋯	1	1	1	0	0	0
R124	0	0	0	0	⋯	0	1	1	1	0	0
R125	0	0	0	0	⋯	0	0	1	1	1	0
R126	0	0	0	0	⋯	0	0	0	1	1	0
R127	0	0	0	0	⋯	0	0	0	0	1	1
R128	0	0	0	0	⋯	1	0	0	0	0	1

Figure 2.5: $\mathbf{Q}^T$ demonstration 128×127

ers i.e., [7] [21] [9] [16] [3] [30] which demonstrate different methods of fault detections. In this section we review briefly the related ones.

In [30], the authors propose fault detection architectures for $GF(2^m)$ multipliers of both bit-parallel and bit-serial types. The polynomial basis is used to represent the field elements. They develop parity prediction schemes for detecting errors due to single and certain multiple faults during the multiplication operation in the field. Fault detection architectures for traditional and bit-parallel multipliers are presented. Explicit formulations for parity predictions of three irreducible polynomials, namely, equally spaced polynomials, trinomials, and pentanomials, are also investigated. Then, the authors have used similar tech-

niques to develop fault detection architectures for both MSB-first and LSB-first bit-serial multipliers. The actual parity of the multiplier output is computed by the Binary Tree of *XOR* (BTX) gates. The predicted parity is calculated based on the coordinates of the input and characteristics of finite field addition and multiplication. Then, the outputs of these two parities are compared to detect the faults. The double parity prediction method is also discussed given the parity of the both inputs to the multiplier is known. The parity prediction block uses both parities and the coordinates of the input to generate the double parity prediction bits that will be compared with the actual parity bits of the multiplier output to indicate Pass or Fail signal.

In [9], the paper provides the concurrent error detection scheme for all-one polynomial to protect the encryption and decryption process against both faults and attacks. To accomplish, The concept of REcomputing with Shifted Operands (RESO) is selected.The RESO scheme employs time redundancy. Assuming function $F(x)$ to be a function unit and the function $G(x)$ are related by be $G^{-1}(F(G(x))) = F(x)$ for all input x. The results of two computations are compared to indicate existence of error. The proposed method needs two additional clock cycles.

In [3], the selected approach is based on the multiple parity bits and its effect on area overhead and error detection probability. The paper discusses the multiple bit errors in bit parallel and bit serial polynomial basis multipliers with respect to selected number of parity bits in error detection scheme. In this

approach, the m bit input is divided into k parts with one parity bit for each, then the multiple parity prediction scheme is used to conduct the comparison of predicted and actual blocks to detect any errors. The author demonstrates that in the bit-serial implementation of $GF(2^{163})$ PB multiplier using 8 parity bits, the area overhead of 10.29% and probability of 0.996 are obtained. This is achieved without increase in the computation time in multiplier.

Chapter 3

Single bit fault detection in GCM

3.1 Fault Detection Scheme

In this chapter, we introduce a novel single-bit parity prediction method in the GCM loop. The parity prediction scheme is outlined and extended to include the coordinates of the Ciphertext rather than the coordinates of the input to the multiplier. The parity prediction scheme is derived using the properties of $GF(2^{128})$ with $f(x) = x^{128} + x^7 + x^2 + x + 1$ as irreducible polynomial. Then, we compare the predicted and actual parities in order to detect the faults in the GCM loop. For the purpose of fault detection in the GCM module, we assume that the AES encryption part of the module is fault free or its fault could be verified by known methods discussed in Chapter 2. Thus, our main concern will be the fault detection in the $GHASH$ function of the GCM module.

As shown in Figure 2.1, the output of the $GHASH$ function is X_i whose parity can be calculated using Binary Tree of XOR (BTX) gates or Linear Feed-

back Shift Register ($LFSR$) to generate p_{X_i}. We need to predict the parity of multiplier output X_i depicted as $\hat{p}_{X_i}$ to compare with the actual parity p_{X_i} in order to generate PASS or FAIL signal as error indication output.

Let's start with the sum module of Figure 2.1. The sum module denoted by $\oplus$ adds two elements in $GF(2^{128})$ which is 128 2-input XOR gates to perform additions over $GF(2)$. The sum result is the bitwise XOR of X_{i-1} and C_i shown as

$$D_i = X_{i-1} + C_i = (x_0 + c_0, \ldots, x_{127} + c_{127}) \in GF(2^{128}). \tag{3.1}$$

Let $p_{X_{i-1}}$ and p_{C_i} denote the parity bit of $X_{i-1} = Y_i$ and C_i respectively. To obtain the parity of the result p_D over $GF(2^{128})$ one can write:

$$p_{D_i} = \sum_{i=0}^{127}(y_i + c_i) = \sum_{i=0}^{127} y_i + \sum_{i=0}^{127} c_i = p_{X_{i-1}} + p_{C_i}. \tag{3.2}$$

Equation (3.2) shows that the parity will be saved in the sum module and remains intact during the operation. The output of sum module denoted by $X_{i-1} + C_i$ is applied to the multiply module depicted as $\otimes$ to perform multiplication by $Hash\ Key\ H$ over $GF(2^{128})$ mod $F(\alpha)$ where α is the root of irreducible polynomial $f(x)$. Next, we figure out the parity prediction method in the multiply module in terms of the GCM elements e.g., C_i and H. The parity of the multiplier output denoted by X_i will be a function of its input operands D_i and H or a function of parity for each operand.

$$p_{X_i} = f(D_i, H) = f(p_{D_i}, p_H). \tag{3.3}$$

This formulation shows that the parity of the output is a function of the parity of input Ciphertext C_i and the parity of $Hash\,Key\,H$.

3.1.1 Parity for the powers of the Hash Key(H)

In this section, we focus on multiplication in $GF(2^{128})$ to implement the parity prediction scheme for the multiplier in the GCM. Authors in [30] have shown the following characteristics of $GF(2^m)$ which we use to outline our fault detection strategy over AES-GCM module.

Let $A \in GF(2^{128})$ and α be a root of irreducible polynomial $f(x)$, then each element of the field could be written in terms of α as

$$A = \sum_{i=0}^{127} a_i \alpha^i \quad a_i \in \{0, 1\} \tag{3.4}$$

Where a_is are the coordinates of A with respect to polynomial basis.

The finite field multiplication of two elements A and B in $GF(2^{128})$ could be represented as

$$A\,mod\,F(\alpha) = A \cdot \sum_{i=0}^{127} b_i \cdot ((A\alpha^i)\,mod\,F(\alpha)) = \sum_{i=0}^{127} b_i \cdot Z^{(i)} \tag{3.5}$$

Where

$$Z^{(i)} = \alpha \cdot Z^{(i-1)} \, mod \, F(\alpha) \;\; for \, 1 \leq i \leq 127, \, Z^{(0)} = A. \tag{3.6}$$

In the AES-GCM module shown in Figure 2.1, the *Hash Key H* is encrypted initially and remains constant throughout the Tag generation process. Using this feature, we propose a parity prediction method in the GCM loop in terms of the parity of $H^{(j)}s$ and parity of Ciphertext C_i in order to have low complexity and space overhead. To Calculate the parity prediction of the multiplier output, we need to compute the parity of $H^{(j)}s$ by using (3.6) as

$$H^{(j)} = \alpha.H^{(j-1)} mod \, F(\alpha) \tag{3.7}$$

$H \, and \, H \cdot \alpha$ could be written in terms of the its coordinate as follows

$$\begin{aligned} H &= h_0 + h_1\alpha + \cdots + h_{127}\alpha^{127} \\ H \cdot \alpha &= h_0\alpha + h_1\alpha^2 + \cdots + h_{126}\alpha^{127} + h_{127}\alpha^{128} \end{aligned} \tag{3.8}$$

Since α is a root of irreducible polynomial and $F(\alpha) = 0$, then α^{128}=$\alpha^7 + \alpha^2 + \alpha + 1$. Therefore, substitution of α^{128} with its equivalent in (3.8) obtains $H \cdot \alpha \, mod \, F(\alpha)$ as

$$H^{(1)} = h_0\alpha + h_1\alpha^2 + \cdots + h_{126}\alpha^{127} + h_{127}(\alpha^7 + \alpha^2 + \alpha + 1) \tag{3.9}$$

Thus, we can write the vector notation of $H \cdot \alpha \, mod \, F(\alpha)$ as

$$\mathbf{h}^{(1)} = [h_{127},\, h_0{+}h_{127},\, h_1{+}h_{127},\, h_2,\, h_3,\, h_4,\, h_5,\, h_6{+}h_{127},\, h_7,\, \ldots,\, h_{126}]^T. \quad (3.10)$$

From (3.10) we conclude the important parity relationship between powers of H as

$$\begin{aligned} p_{H^{(1)}} &= p_{H^{(0)}} + h_{127} \\ p_{H^{(2)}} &= p_{H^{(1)}} + h_{126} \\ &\vdots \\ p_{H^{(127)}} &= p_{H^{(126)}} + h_1. \end{aligned} \quad (3.11)$$

3.1.2 Parity Prediction for the Multiplier in the GCM

The authors of [30] have derived the parity prediction formula for the multiplication of two arbitrary field elements which can be applied to the multiplier inputs D_i and H in the AES-GCM module as follows

$$\hat{p}_{X_i} = \sum_{i=0}^{127} d_i p_{X^{(i)}} = \sum_{i=0}^{127} d_i p_{H^{(i)}}. \quad (3.12)$$

The formulation of (3.12) constructs the foundation in parity prediction of the AES-GCM module in this thesis. As the $Hash\,Key\,H$ does not change during the encryption process, the values of $p_{H^{(i)}}$ could be precomputed and stored in register PH at the beginning of each encryption. Then, we obtain bitwise AND of PH register with D_i.

Next, we calculate the predicted parity by performing *XOR* operation on outputs as shown in Figure 3.1. The predicted parity can be compared with the actual parity of the output to verify any odd numbers of stuck at-0 or stuck at-1 fault happened in the circuit under test (CUT).

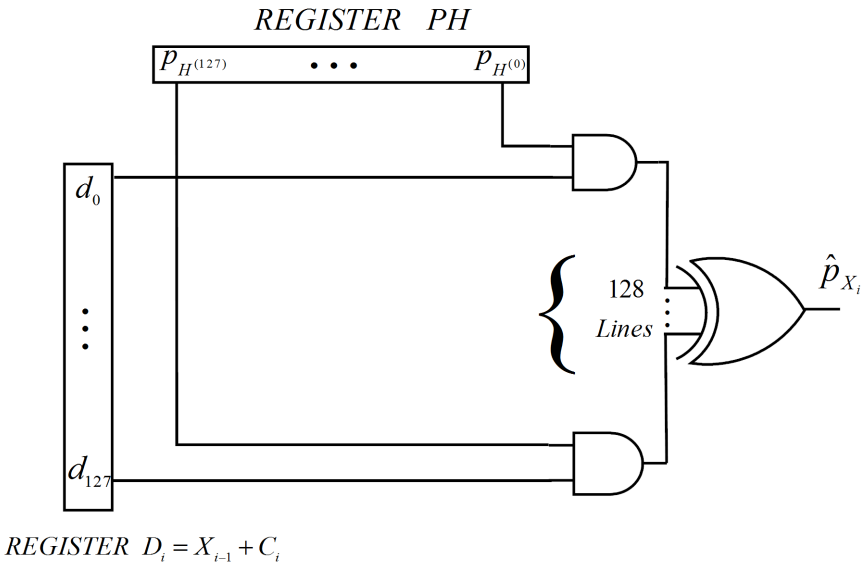

Figure 3.1: Parity prediction scheme

3.1.3 Fault Detection in the GCM Loop

We need to extend fault detection to include the parity of the actual Ciphertext. Therefore, we rewrite D_i in terms of the inputs of sum module. Thus, we can rewrite (3.12) at the *ith* clock cycle as follows:

$$X_i = (X_{i-1} + C_i) \odot p_H = (Y_{i-1} \odot p_H) + (C_i \odot p_H). \tag{3.13}$$

where operator $\odot$ denotes bitwise AND operation. Let Y_{i-1} and C_i represent

the contents of Y register and Ciphertext C_i at the *ith* clock cycle respectively. Therefore, the parity prediction of the $GHASH$ function at the *ith* clock cycle in the AES-GCM loop can be found using (3.14) as follows

$$\hat{p}_{X_i} = \sum_{j=0}^{127} y_j p_{H^{(j)}} + \sum_{j=0}^{127} c_j p_{H^{(j)}}. \tag{3.14}$$

where $Y_{i-1} = (y_0, \ldots, y_{127})$ and $C_i = (c_0, \ldots, c_{127})$. Figure 3.2 shows the AES-GCM parity prediction scheme which is a realization of the key formulation presented in (3.14).

The error indicator e_{out} is generated using one XOR gate as shown in Figure 3.3. At every clock cycle the existence of error will be verified by detecting logic 1 at e_{out}. All needed at the output of the multiplier is to compute its parity p_{X_i} using the BTX.

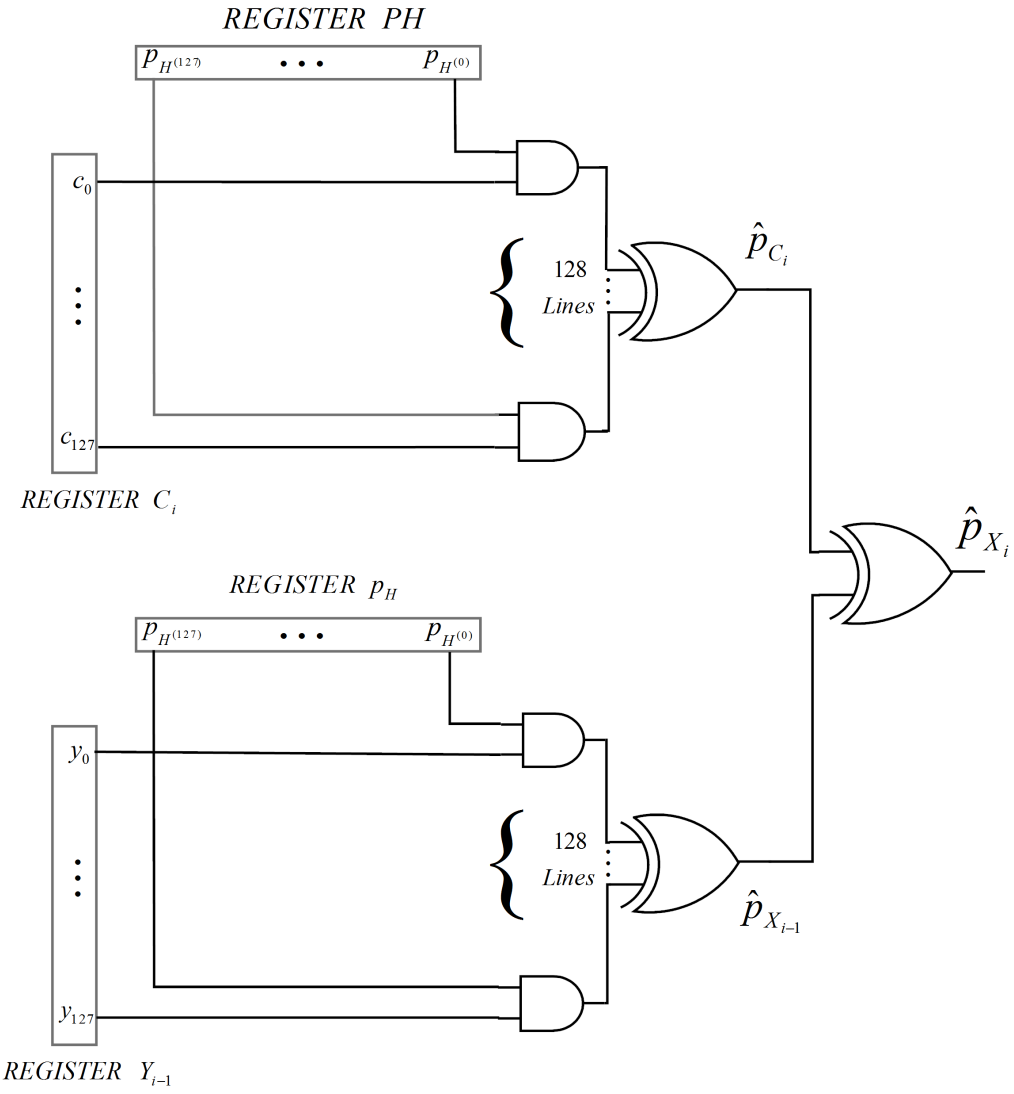

Figure 3.2: AES-GCM loop parity prediction scheme

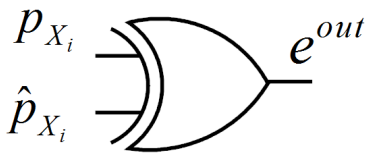

Figure 3.3: Output error indicator

Chapter 4

Multiple Parity Bit Fault Detection Architecture

4.1 Introduction

In this chapter we introduce a novel matrix-based multiple parity bit fault detection method in the AES-GCM loop. As shown in [31] and discussed in Section 2.4.1, the multiplier output X_i in Figure 2.1 can be represented as $\mathbf{Ed} = (\mathbf{L} + \mathbf{Q}^T\mathbf{U})\mathbf{d}$, where $\mathbf{E}$ and $\mathbf{d}$ represent the H and D_i inputs to the multiplier respectively.

First, we define the matrix $\mathbf{E}$ in terms of H and outline single and multiple bit parity fault detection. Next, matrix-based *CRC* is introduced and related fault detection scheme is investigated.

4.2 Matrix-Based Parity Prediction Scheme in GCM Loop

4.2.1 Matrix-Based Single Parity Bit Scheme

In this section, we define the matrix- based parity prediction scheme for the multiplier output. As the parity of the multiplier output $X_i = C$ is defined by $p_C = \sum_{i=0}^{127} c_i$, we show the matrix presentation of the output parity as

$$p_C = \mathbf{oc}, \tag{4.1}$$

where $\mathbf{c} = [\, c_0 \quad c_1 \quad \dots \quad c_{127} \,]^T$ and $\mathbf{o} = [\, 1 \quad 1 \quad \cdots \quad 1]$ is a 1×128 all one row vector. For the purpose of output parity prediction, we need to use the coordinates of the inputs to the multiplier which can be obtained from matrix **E** as follows

$$\hat{p}_C = (\mathbf{oE})\mathbf{d}, \tag{4.2}$$

where $\mathbf{d} = [\, d_0 \quad d_1 \quad \cdots \quad d_{127}]^T$, **oE** is a function of H which is denoted as $[\, 1 \quad 1 \quad \cdots \quad 1][\, f(H)]_{128\times128}$. Thus, the parity prediction of the output will be function of H and D_i, i.e,

$$\hat{p}_C = f\,(H, D_i), \tag{4.3}$$

To outline the parity prediction architecture, we focus on computing the outcome of **oE** instead of finding matrix **E**. Therefore, our new approach is to obtain the parity of each column of **oE** instead of computing the actual multi-

plication.

The Matrix $\mathbf{Q}^T$ shown in Figure 2.4 is multiplied by $\mathbf{U}$ in order to obtain $\mathbf{Q}^T\mathbf{U}$. Let the number of $1s$ in the jth column of $\mathbf{Q}^T$ be r_j. Then each element of $\mathbf{U}$ in the jth row repeats r_j times in each column of $\mathbf{Q}^T\mathbf{U}$. For example, the first column of $\mathbf{Q}^T$ contains 4 $Ones(1s)$ which causes each element of $\mathbf{U}$ in the first row to appear 4 times in each column of $\mathbf{Q}^T\mathbf{U}$. The first column of $\mathbf{U}$ is all $0s$ which does not have any effect once added to the first column of $\mathbf{L}$. Thus, the content of the first element of $\mathbf{oE}$ denoted by $\mathbf{oE_{(1,1)}}$ is the parity of the first column of $\mathbf{L}$ that is p_H.

The second column of $\mathbf{U}$ has h_{127} in the first row and all $0s$ for the rest of the column. Therefore, h_{127} appears four times in the second column of $\mathbf{Q}^T\mathbf{U}$ which has no effect on the parity of the second column of $\mathbf{Q}^T\mathbf{U}$ because of the even numbers of the repeats of h_{127} that cancels off the effect. The value of the second element of the $\mathbf{oE}$ denoted by $\mathbf{oE_{(1,2)}}$ is the parity of the second column of $\mathbf{L}$ which is $p_H + h_{127}$. This pattern continues up to the element 122 with the corresponding $\mathbf{oE_{(1,122)}} = \mathbf{oE_{(1,121)}} + h_7$.

The pattern changes from the element 123 shown as

$$
\begin{aligned}
\mathbf{oE_{(1,1)}} &= p_H \\
\mathbf{oE_{(1,j)}} &= \mathbf{oE_{(1,j-1)}} + h_{m-(\mathbf{j}-1)} \qquad for\, 2 \leq j \leq 122 \\
\mathbf{oE_{(1,123)}} &= h_0 + h_1 + h_2 + h_3 + h_4 + h_5 + h_{127} \\
\mathbf{oE_{(1,124)}} &= h_0 + h_1 + h_2 + h_3 + h_4 + h_{126} + h_{127} \\
\mathbf{oE_{(1,125)}} &= h_0 + h_1 + h_2 + h_3 + h_{125} + h_{126} + h_{127} \\
\mathbf{oE_{(1,126)}} &= h_0 + h_1 + h_2 + h_{124} + h_{125} + h_{126} + h_{127} \\
\mathbf{oE_{(1,127)}} &= h_0 + h_1 + h_{123} + h_{124} + h_{125} + h_{126} + h_{127} \\
\mathbf{oE_{(1,128)}} &= h_0 + h_{122} + h_{123} + h_{124} + h_{125} + h_{126},
\end{aligned}
\tag{4.4}
$$

where $\mathbf{oE_{(1,k)}}$ contains the $p_{H^{(k)}}$. Figure 4.1 shows the logic implementation of the matrix **oE** which is stored into the register **oE**.

The register **oE** can be used for single parity prediction and fault detection in the AES-GCM loop using the same architecture shown in Figure 3.2. Thus, the register **oE** replaces register p_H while the rest of the scheme remains the same for the purpose of fault detection.

4.2.2 Matrix-Based Random Parity Bit Scheme

In this section, we extend the parity prediction of (4.2) to double and then to multiple parity bit scheme for the purpose of fault detection. Therefore, we change the matrix **o** from 1×128 to $\mathbf{O}'$ with a new dimension of $k \times 128$ to

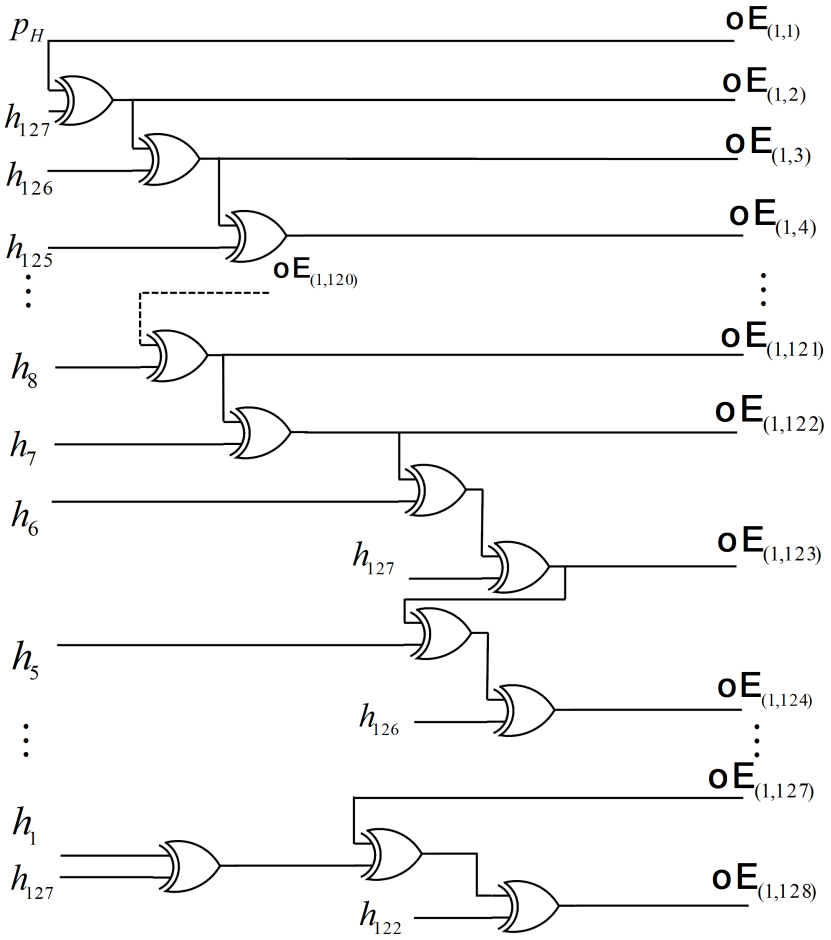

Figure 4.1: Implementation of Matrix(Register) **oE**

obtain k parity bits in our scheme as follows

$$\begin{bmatrix} \hat{p}_{C_1} \\ \hat{p}_{C_2} \\ \vdots \\ \hat{p}_{C_k} \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & \cdots & 0 \\ 1 & \cdots & 0 \\ \vdots & \cdots & \vdots \\ 0 & \cdots & 1 \end{bmatrix}}_{\mathbf{O}'(k\times 128)} (\mathbf{E}.\mathbf{d}) \tag{4.5}$$

where $\mathbf{O}'$ can be a random matrix. The selected pattern of $\mathbf{O}'$ depends on complexity and fault coverage in the fault detection module. Double parity prediction block diagram is shown in Figure 4.2.

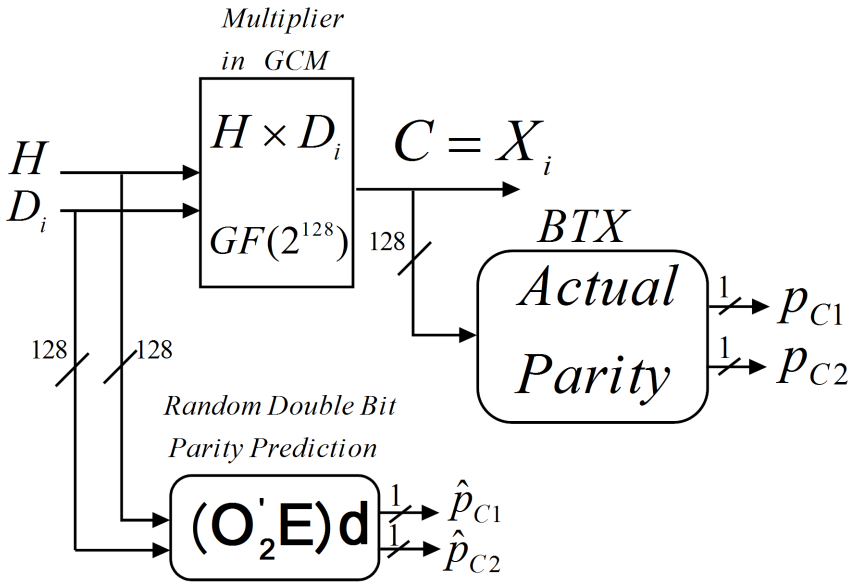

Figure 4.2: Block Diagram of Double Bit Parity.

4.3 Matrix-based CRC for Multi-Bit Parity Fault Detection

4.3.1 Brief review on CRC

CRC (Cyclic Redundancy Check) [8] [32], is a code for detecting errors in digital networks, data transmission, and storage devices. *CRC* has been adopted to detect changes to data, but no error correction is made through it. In the *CRC*, certain number of bits will be added to the message that is called checksum to be transmitted together with the message. Then, the checksum of the received message is computed and compared with the sent one to detect possible errors occurred during the transmission.

CRC treats the message as a polynomial in $GF(2^m)$. It is obtained by computing the remainder of dividing the message polynomial into the divisor (generator) polynomial. For $k-$bit *CRC*, the generator polynomial must be of degree k. Let denote the message as $m(x) = m_{m-1}x^{n-1} + \cdots + m_1x + m_0$, the generator polynomial as $G = g(x) = g_kx^k + \cdots + g_1x + 1$; Then, we outline the *CRC* of $m(x)$ as follows:

$$m(x)\, mod\, g(x) \; = CRC(m). \tag{4.6}$$

For $k-$bit *CRC*, the check value is k bits and the generator polynomial has $(k + 1)$ terms. A $k-$bit *CRC* is capable of detecting all errors of length $\leq k$. If G is $x + 1$, then the *CRC* denoted as $CRC - 1$ is called the parity bit to detect single bit or odd number of errors. Most commonly used CRCs are CRC-12, CRC-16, and CRC-32.

4.3.2 Matrix-Based Double Bit Parity CRC

In this section, we define the matrix-based *CRC* fault detection scheme using the multiple parity bits for the multiplier module. Then, we extend the proposed fault detection to the entire GCM loop. The multiplier output $c(x)$ is a

polynomial of degree 127 as shown below:

$$\mathbf{c}^T\mathbf{x} = c(x)$$

$$\begin{bmatrix} c_0 & c_1 & \dots & c_{127} \end{bmatrix}^T \begin{bmatrix} 1 \\ x \\ \vdots \\ x^{127} \end{bmatrix} = c(x), \tag{4.7}$$

where $c(x) = c_0 + c_1x + \ldots + c_{127}x^{127}$. For the purpose of double parity prediction scheme, the generator polynomial is $g(x) = x^2 + x + 1$. As the degree of the generator polynomial is 2, the resulting remainder would be $ax + b$ which creates two bit parity for the purpose of fault detection. From (4.7), we realize that the coordinates of the multiplier output $c(x)$ can affect the output parities after calculation of $\mathbf{x}\ mod\ g(x)$ which is independent of the output coordinates and can be used as constant pattern for parity generation and prediction. Thus, we conclude the following equation

$$c(x)\ mod\ g(x) = \mathbf{c}^T.(\mathbf{x}\ mod\ g(x)) \tag{4.8}$$

Now, we compute the $\mathbf{x}\ mod\ g(x)$ and establish the related matrix as

$$\begin{bmatrix} 1 \\ x \\ x^2 \\ x^3 \\ x^4 \\ \vdots \end{bmatrix} \; mod \; g(x) = \begin{bmatrix} 1 \\ x \\ x+1 \\ 1 \\ x \\ \vdots \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ 1 & 0 \\ 0 & 1 \\ \vdots & \vdots \end{bmatrix} \begin{bmatrix} 1 \\ x \end{bmatrix} = \mathbf{p}_{CRC} \begin{bmatrix} 1 \\ x \end{bmatrix} \tag{4.9}$$

Using (4.9), we realize a very important approach towards the proposed fault detection by introducing the *CRC* of double bit parity matrix as $\mathbf{p}_{CRC-2}$. Thus, form (4.9) and (4.8) the double bit parity of the output is computed as follows

$$\mathbf{c}^T.\mathbf{p}_{CRC-2} \begin{bmatrix} 1 \\ x \end{bmatrix} = [b \quad ax] = [p_{C1} \quad p_{C2}] \begin{bmatrix} 1 \\ x \end{bmatrix} \qquad a, b \in \{0, 1\} \tag{4.10}$$

To calculate the $[p_{C1} \quad p_{C2}]$, one should multiply $\mathbf{c}^T$ by each column of $\mathbf{p}_{CRC-2}$ separately. The hardware realization of this operation is bitwise *AND* of multiplier output $\mathbf{c}^T$ with column 1 and 2 of $\mathbf{p}_{CRC-2}$. Then, we compute each individual parity by performing *XOR* operations on the output bits to obtain p_{C1} and p_{C2} respectively as shown in Figure 4.3.

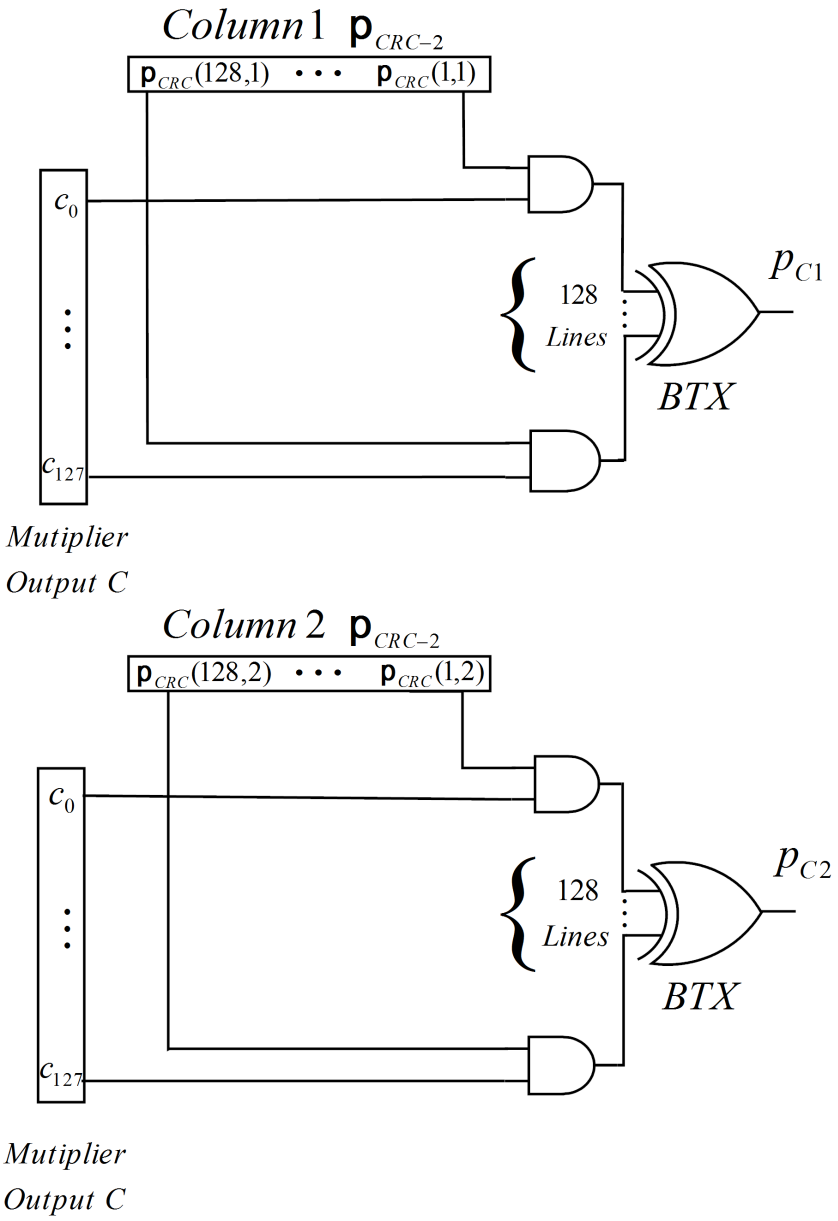

Figure 4.3: Hardware implementation of Double Bit Parity Generator on multiplier output.

4.3.3 Matrix-Based Double Bit Parity Prediction CRC

In this section, we outline the parity prediction scheme using the matrix-based *CRC*. Figure 4.1 demonstrates the content of matrix **OE** which is absolutely a function of H. Therefore, matrix $\mathbf{O}'\mathbf{E}$ is a function of H as well.

As shown in Figure 4.5, the inputs to the GCM multiplier are H and D_i. Thus, matrix $\mathbf{O}_k^{'}\mathbf{E}$ represents k bit parity prediction with regards to the multiplier inputs H and D_i which is denoted by $d(x)$ in polynomial basis. Using (4.9), we define the output parity prediction as follows

$$(\mathbf{O}_k^{'}\mathbf{E})\mathbf{d} = \hat{\mathbf{p}}_{(CRC-k)}\mathbf{d}$$
$$\mathbf{d} = \begin{bmatrix} d_0 \\ d_1 \\ \vdots \\ d_{127} \end{bmatrix}, \tag{4.11}$$

where $\hat{\mathbf{p}}_{(CRC-k)}\mathbf{d} = [\hat{p_0} \quad \hat{p_1} \quad \cdots \quad \hat{p_k}]^T$. As we are investigating double parity bit fault detection, we replace the $\mathbf{O}_k^{'}$ with $\mathbf{p}_{(CRC-2)}^T$. Thus, we use (4.11) to obtain the double bit parity prediction as follows

$$(\mathbf{p}_{(CRC-2)}^T\mathbf{E})\mathbf{d} = [\hat{p}_{C1} \quad \hat{p}_{C2}]. \tag{4.12}$$

The computation of $\mathbf{p}_{(CRC-2)}^T\mathbf{E}$ is given in appendix A which could be easily implemented in hardware. In order to detect faults, we need to compare the actual and the predicted parities to generate PASS or FAIL signal which is called e^{out}. If the compared parity pairs are the same, there is no error and $e^{out} = 0$, otherwise an error signal is generated and $e^{out} = 1$ as shown in Figure 4.4

which is the "$k - bit\ XOR\ Comparator$" block depicted in Figure 4.5.

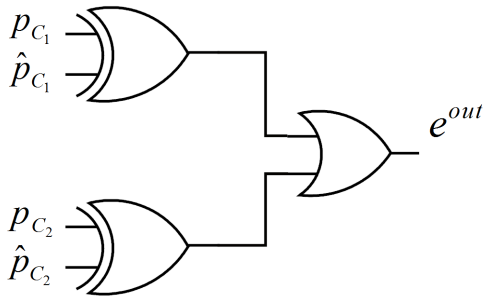

Figure 4.4: Error signal generator.

4.3.4 Matrix-based k Bit Parity Fault Detection ($k > 2$)

For the k bit matrix-based fault detection scheme, one needs to define the generator polynomial which is degree of k. The generator polynomial can be irreducible which gives the maximum number of checksums or remainders. Therefore, we have the maximum code length of $2^k - 1$ and the code can detect all one bit and double bit errors [18]. To improve the error detection capability, we choose the generator polynomial to be $g(x) = (x + 1)g'(x)$ where $g'(x)$ is primitive polynomial. The order of $g'(x)$ is $k - 1$ and total code length will be $2^{k-1} - 1$, in this case all single, double, triple, and all odd number of errors could be detected [36]. Table 4.1 illustrates the irreducible polynomials for the *CRC* up to the degree 5. Higher degree polynomials can be found in [18].

After selection of the generator polynomial, we need to determine the pattern of matrix $\mathbf{p}_{CRC-k}$ with regards to chosen polynomial. To accomplish, we calculate the set $\{1, x, x^2, \ldots, x^{127}\}\ mod\ g(x)$ which is called $r_j s$ defined as

k	*Irreducible Polynomial*
1	$x+1$, x
2	x^2+x+1
3	x^3+x^2+1, x^3+x+1
4	$x^4+x^3+x^2+x+1$, x^4+x^3+1, x^4+x+1
5	$x^5+x^4+x^3+x^2+1$, $x^5+x^4+x^3+x+1$, $x^5+x^3+x^2+x+1$, $x^5+x^4+x^2+x+1$, x^5+x^3+1, x^5+x^2+1

Table 4.1: Irreducible Polynomials to the degree 5.

$r_j = x^j \, mod \, g(x)$ for $0 \leq j \leq 127$. Then, the remainder set $\{r_1, r_2, \ldots, r_{128}\}$ constructs the rows 1 *to* 128 of the parity matrix $\mathbf{p}_{CRC}$ respectively. Table 4.2 shows the pattern for selected polynomials.

Polynomial	x^2+x+1	x^3+x+1	x^4+x+1
$\mathbf{p}_{CRC}$	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 1 \\ \vdots & \vdots \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \\ \vdots & \vdots & \vdots \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$

Table 4.2: $\mathbf{p}_{CRC}$ Matrix pattern for selected polynomials.

4.3.5 Fault Detection Architecture

At this stage, we load each column of $\mathbf{p}_{CRC-k}$ matrix into k separate registers called $\mathbf{p}(j)_{CRC-k}$ where j indicates the jth column and $1 \leq j \leq k$. $\mathbf{p}_{CRC-k}$ does not change in GCM encryption process which can be computed and stored

permanently in the GCM module. Then, by implementing (4.10) and (4.8), we figure out the actual parities and predicted k- bit parities to compare and generate e^{out} signal.

To extend the fault detection to the entire GCM loop, we outline the block diagram of the model using $k-$bit parity prediction scheme in Figure 4.5. In this scheme, we take into account the coordinates of the Ciphertext which is the main element of the GCM module in the $GHASH$ function. As illustrated in Figure 4.5, register C_i represents the coordinates of the *ith* Ciphertext, register X_i is the current multiplier output, register Y_{i-1} is the content of $X_{i-1} + C_i$. The signal e^{out} indicates presence of fault in each clock cycle.

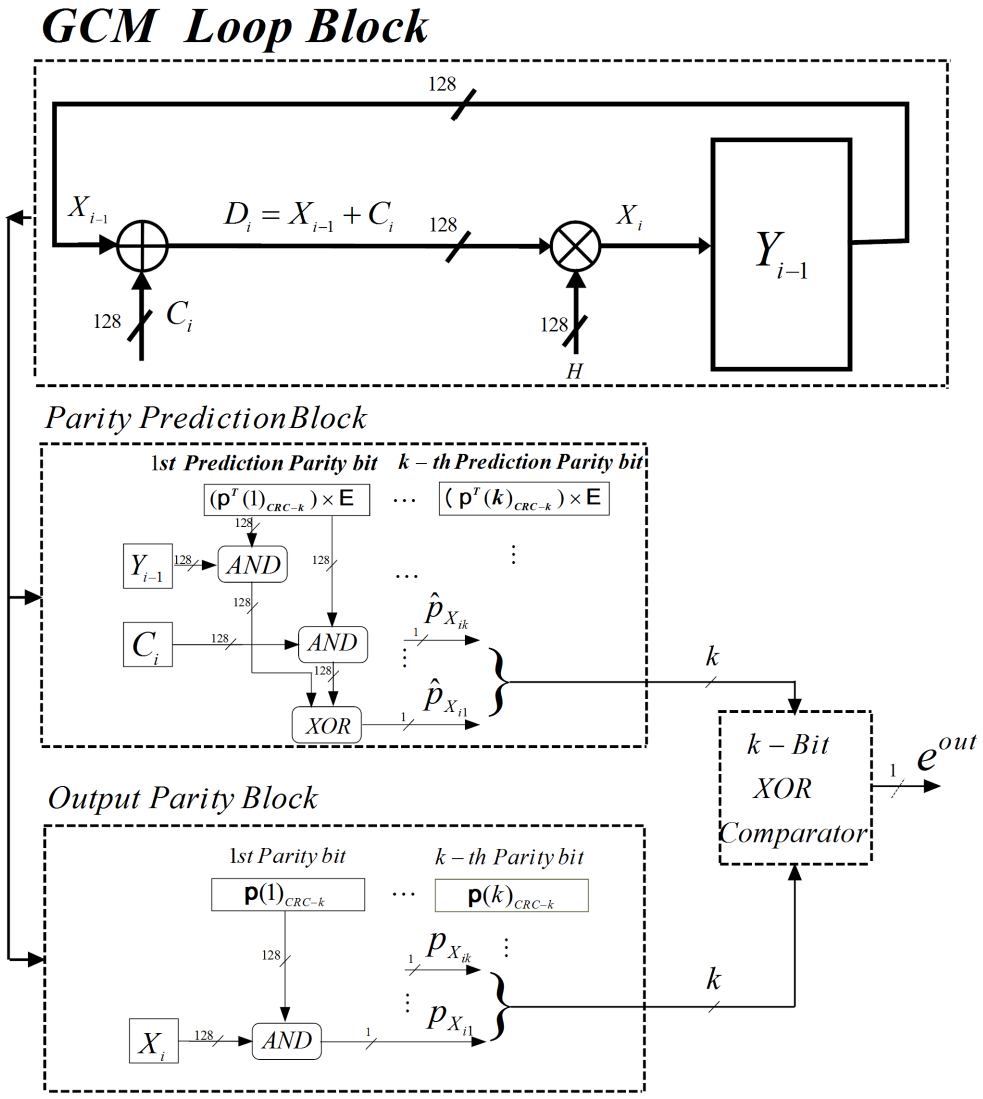

Figure 4.5: Block diagram of $k-$bit parity fault detection in GCM loop.

Chapter 5

Testing and Simulation

5.1 Introduction

Since we have used the information redundancy technique in our fault detection scheme, a parity based Concurrent Fault Detection (CFD) has been investigated. In this chapter, we introduce the simulation testbench for the proposed matrix-based multiple bit parity prediction *CRC* discussed in Chapter 4. First, we implement the GCM scheme using VHDL language. Then, Modelsim SE 10.3 is applied to simulate the design. The fault injection is implemented in the body of the main program using the VHDL language. TCL stimulus package is employed to inject the faults and monitor the outputs. The different instances of single, multiple bit, transient, and permanent stuck-at faults are injected into the scheme and the results are investigated and further elaborated in the following sections.

5.2 VHDL Implementation of Fault Model

The main part of the GCM is multiplier module which is implemented through equation $C = (\mathbf{Q}^T\mathbf{U} + \mathbf{L})$ [31]. We can write the multiplier output of the GCM module shown in Figure 4.5 as

$$\mathbf{x_i} = (\mathbf{L} + \mathbf{Q}^T\mathbf{U})\mathbf{d_i} \tag{5.1}$$

All elements of $\mathbf{Q}^T\mathbf{U}$ and $\mathbf{L}$ are calculated and constructed in terms of the coefficients of H. Matrix $\mathbf{Q}^T\mathbf{U}$ is depicted in Figure 5.1. Matrix $\mathbf{E}$ represents $(\mathbf{L} + \mathbf{Q}^T\mathbf{U})$ which generates the multiplier output by performing multiplication $\mathbf{E}.\mathbf{d_i}$ over $GF(2^{128})$ where $\mathbf{d_i} = (\mathbf{c_i} + \mathbf{x_{i-1}})$ which are discussed in Chapter 2. Therefore, we need to define all the matrices in terms of the VHDL language which is demonstrated in appendix A.

For the purpose of fault injection and fault detection, the implementation of all matrices and vectors must be done individually in VHDL to give us the gate level access for the GCM module. Otherwise, we will not be able to cover entire hardware to inject the faults. To accomplish this approach, we introduce and add $Element\_Stuck\_at\_fault$ to all elements of each matrix and vector separately as follows

$$(Element(m,n)\ AND\ Element\_Stuck\_at\_0(m,n))\ OR\ Element\_Stuck\_at\_1(m,n), \tag{5.2}$$

where $Element(m,n)$ represents each vector or matrix defined in (5.1). Associated Stuck-at-0 and Stuck-at-1 matrices are defined through the other introduced matrices $Element\_Stuck\_at\_0$ and $Element\_Stuck\_at\_1$ respectively. $Element\_Stuck\_at\_0$ and $Element\_Stuck\_at\_1$ are initialized to all $1s$ and all $0s$ subsequently. To switch on the fault injection in the $Element(m,n)$, we need to change $Element\_Stuck\_at\_0(m,n)$ to logic 0 or $Element\_Stuck\_at\_1$ to logic 1 for the period of multiplication operation.

To continue to the next fault evaluation, we need to restore the injected fault into the $Element\_Stuck\_at\_fault$ to the initial no fault values in order to suppress the effect of other faults on the current one. The flowchart of fault injection method used in the proposed scheme is shown in Figure 5.2.

The first step is to initialize all the $Element\_Stuck\_at\_fault$ matrices to proper logic 1 and 0 which do not perform any fault injection to the any *Element* matrix. Then, for each element of this matrix, one can inject Stuck-at-0 by setting $Element\_Stuck\_at\_0(m,n)$ to logic 0. Next, the output of the GCM loop is checked to detect the effect of the fault. We repeat the fault injection process for $Element\_Stuck\_at\_1(m,n)$ respectively. All the elements of matrices are tested against different faults and the results are captured for coverage calculations.

As an example, we show how to inject the fault in element$(1,2) = h_{127}$ of the $\mathbf{Q}^T\mathbf{U}$ matrix illustrated in Figure 5.1. We have specified the corresponding fault injection stuck_at_0 and stuck_at_1 matrices in the VHDL implementation as USA_0 and USA_1 respectively, Thus, the presentation of the fault at element

$\mathbf{Q}^T\mathbf{U}(1,2)$ is as follows

$$\mathbf{Q}^T\mathbf{U}(1,2) = (h_{127}\ AND\ USA\_0(1,2))\ OR\ USA\_1(1,2) \tag{5.3}$$

To inject stuck_at_0 respectively stuck_at_1 for this element, we need to force USA_0 to logic 0 respectively USA_1 to logic 1. This is done through the scripting language explained in following sections. We repeat the process for other matrices e.g., $\mathbf{L}$, and $\mathbf{E}$. For each element of $\mathbf{Q}^T\mathbf{U}$ matrix which contains more than one coordinates, we introduce two vectors named $UUSA\_0(l)$ and $UUSA\_1(l)$. $UUSA\_0(0)$ and $UUSA\_1(0)$ are inserted at $\mathbf{Q}^T\mathbf{U}(1,123)$ which has two coordinates of H. From Figure 5.1, we illustrate the fault injection at gate level of $\mathbf{Q}^T\mathbf{U}(1,123) = h_6 + h_{127}$ as follws

$$\begin{aligned}\mathbf{Q}^T\mathbf{U}(1,123) = (((h_6\ AND\ UUSA\_0(0))\ OR\ UUSA\_1(0)) \\ + ((h_{127}\ AND\ UUSA\_0(1))\ OR\ UUSA\_1(1))) \\ AND\ USA\_0(1,123))\ OR\ USA\_1(1,123),\end{aligned} \tag{5.4}$$

The gate level illustration of (5.4) is shown in Figure 5.3.

After injection of faults for all matrices forming $\mathbf{E}$ in (5.2), the next step is to inject fault for the other multiplier input $\mathbf{d_i}$ in $\mathbf{E}.\mathbf{d_i}$ which leads to multiplication result X_i. As $\mathbf{E}$ is 128 *by* 128 matrix, all coordinates of d appear in all coordinates of output $C = X_i$. Thus, we need to define two matrices $DSA\_0(128,128)$ and $DSA\_1(128,128)$ to calculate the effect of fault injection into each coordinates of d. $DSA\_0(1,1)$ to $DSA\_0(1,128)$ inject stuck_at_0 fault for d_0 when

calculating c_0 to c_{127}. Respectively, $DSA\_0(2,1)$ to $DSA\_0(2,128)$ inject faults for d_1 and so on. For multiplier output $C = X_i$, we appoint $CSA\_0(128)$ and $CSA\_1(128)$ to activate the fault injection .

5.3 Fault Injection in the GCM loop

As shown in Figure 5.4, the output of the multiplier is stored in register Y_{i-1}. Therefore, in the *ith* clock cycle we add the Ciphertext C_i with Y_{i-1}. Then, we apply $(C_i + Y_{i-1})$ to the multiplier input to calculate $(C_i + Y_{i-1}).H$. Thus, we need to take into account the faults in register Y, C_i, and addition (XOR) operation of these two. We need to define related fault injection vectors in the VHDL program which are YSA_0, YSA_1, CSA_0, *and* CSA_1. To simulate the fault injection, we define two registers to replace register Y as $Y1$ which is applied to the main GCM loop calculations and $Y2$ which takes part in parity prediction calculations. Therefore, one can inject the faults into the $Y1$ through Tcl commands while the parity prediction part $Y2$ remains intact. The same model is applied to C_i for the purpose of fault injection into the coordinates of Ciphertext. The effect of fault injection on the fault coverage of the entire GCM loop is discussed in the following sections.

5.4 Simulation Results

To evaluate the error detection capability of the proposed matrix-based parity prediction scheme, the simulation is performed using ModelSim SE 10.3. The

different cases of single and multiple bit faults are injected into the multiplier module, Ciphertext coordinates, register, and adder in the scheme. The fault injection is performed in different sections of the GCM loop using Tcl (Tool Command Language) [2] which is programming/scripting language based on concepts of Lisp, C, and Unix shells. Tcl can be used interactively, or by running package based scripts to get the maximum performance with small number of instructions. In Tcl we can change the value of the signals defined in the main VHDL program. An example is given in (5.5) to change $USA\_0(128, 128)$ to logic 0.

The command that injects $Stuck\_at\_0$ fault to $\mathbf{Q}^T\mathbf{U}(128, 128)$ is as follows

$$Tcl\ command\ to\ set\ signal\ to\ 0 = force\ -freeze\ USA\_0(128, 128)\ 0 \tag{5.5}$$

Enforcing the changes will take place by *run* command for specified time. Then, we are able to monitor the waves (signals) already added to program using $-add\ wave$ command. After running the simulation and examining the waves, we can set back the injected fault to initial no_fault value and continue to the next fault injection as depicted in Figure 5.2. For investigating the fault coverage with regards to the selected parity bits, we first add the signals needed to be monitored, then single or multiple $Stuck\_at\_faults$ are injected into the design entity for run time period. Finally, the error indicator signal is tested to verify the fault coverage.

To switch from interactive to automated testing, we use $for\ loop$ command in Tcl to inject and repeat all types of faults, all the results will be recorded in a

file for further calculations. For the purpose of transient or permanent fault injection, we can control the duration of the injected fault. For permanent faults, the injected fault is not set back to original no_fault state during the entire testing period. To control the duration of the faults in Tcl, we run the program for specified period of time and examine the output waves or other signals using the $RUN\,[time\,ns]$. The sample of Tcl-based fault injection is given in Appendix B. Another method of fault injection and testing is shown in Figure 5.5 which illustrates the simulation result to check the $Stuck\_at\_0$ for Hash Key H input. In the VHDL program We have separated the H input to the multiplier module and the parity prediction module in two different vectors called h, $h1$. The vectors h and $h1$ are applied to multiplier input and the parity prediction input separately. In the Tcl language we force the signals to the desired values, run the simulation for 500 and check the error indicator output. For the first period of $250\,ns$ and the Clock rate of 50, we set $h = h1 =$ "0X...11" which is 128 bit stream formatted in hexadecimal, and examine the error indicator ($EOUT$) which is 0 that means no error signal. Whereas, for the second period of $250\,ns$, with assumption of $h =$ "0X...01" and $h1 =$ "0X...11", the $EOUT$ will change to logic 1 indicating that the fault is detected.

The experimental results for different parity bit implementation and percentage of fault coverage are shown in Table 5.1. Selecting more than 4 parity bits on fault detection will result more than 92% in fault coverage. By choosing 6 parity bits, we achieve 98% in fault coverage.

CRC Parity Bits	Number of fault injections	Fault Coverage
Single bit Parity	300000	48%
Double bit Parity	300000	74%
3 Bit Parity	300000	87%
4 Bit Parity	300000	92.5%
5 Bit Parity	300000	96%
6 Bit Parity	300000	98%

Table 5.1: Fault coverage in the GCM loop versus selected parity bits.

5.5 Fault Detection Overhead and Delay Analysis

For the purpose of evaluating overhead and delay analysis, we selected Altera's Arria VGZ device which offers lower power and higher bandwidth compared to other 28 nm FPGA devices for running applications upto $10Gbps$. Therefore, the overhead and timing analysis was performed by $Quartus \parallel -64\,Bit$ [10] . Table 5.2 shows the effect of adding multiple bit parity fault detection scheme into the CUT versus operating frequency and overhead of the fault free module.

GCM implementation	Overhead in $ALMs$	Longest Delay Path ns
Original :No fault Detection	3445(0%)	2.856
Single Parity	3466(2.8%)	3.387(18.5%)
Double Bit Parity	3665(6.3%)	3.472(21.5%)
3 Bit Parity	3672(6.6%)	3.482(21.9%)
4 Bit Parity	3805(10%)	3.493(22.3%)
5 Bit Parity	3814(10.7%)	3.506(22.7%)
6 Bit Parity	3823(10.9%)	3.515(23%)

Table 5.2: Area overhead and delay versus selected parity bits.

Probability of Fault Coverage for k-bit Parity

The authors in [17] show that the probability of fault coverage using k-bit parity is calculated as $1 - 2^{-k}$. Therefore, the corresponding theoretical outcome for

$k = 1\,to\,K = 6$ are $50\%, 75\%, 87.5\%, 93.7\%, 96.8\%,\ and\ 98.4\%$ respectively. Thus, the simulation results depicted in Table 5.1 with a good approximation, supports the theoretical result discussed in this section. The proposed parity-based fault detection scheme is capable of detecting almost all the injected random faults.

Gate Level Complexity Analysis

Table 5.3 shows the area and critical path delay of the GCM fault detection scheme versus the number of parity bits at gate level count. In order to obtain the overhead percentage in the ASIC design, this analysis helps to provide exact area and the cost of the parity prediction scheme. To obtain the chip area, we take into account that the 2-input AND and 2-input XOR can be built using 6 and 10 transistors respectively.

CRC Parity Bits	Area Complexity	Critical Path Delay
Single bit Parity	$399X + 128A$	$11T_X + T_A$
Double bit Parity	$780X + 512A$	$14T_X + T_A + T_O$
3 Bit Parity	$1382X + 768A$	$19T_X + T_A + 2T_O$
4 Bit Parity	$2237X + 1024A$	$27T_X + T_A + 2T_O$
5 Bit Parity	$4156X + 1280A$	$30T_X + T_A + 3T_O$
6 Bit Parity	$8097X + 1536A$	$39T_X + T_A + 3T_O$

Table 5.3: Gate level area overhead and delay versus selected parity bits, where T_X, T_A, and T_O are the propagation delays of XOR, AND, and OR Gate respectively.

False Alarms

The false alarms are the faults that do not cause any change in output or the parity bits but are alarmed erroneous by the fault detection scheme. The false

alarms is at most 0.12%, which can be ignored with respect to the total number of fault injection that is 300000. Table 5.4 depicts the percentage of false alarm obtained from simulation.

CRC Parity Bits	Number of fault injections	False Alarm
Single bit Parity	300000	270(0.09%)
Double bit Parity	300000	298(0.1%)
3 Bit Parity	300000	300(0.1%)
4 Bit Parity	300000	328(0.11%)
5 Bit Parity	300000	330(0.11%)
6 Bit Parity	300000	360(0.12%)

Table 5.4: False Alarm in the GCM loop versus selected parity bits.

Figure 5.6 shows the the graph of the simulation results.

5.6 Future Work

A future research can be based on obtaining the best CRC generator polynomials in terms of area and timing overhead. Since the power consumption is one of the main factors in each design, implementation of the fault detection scheme into other FPGA devices or ASIC can also be investigated to achieve lower area complexity, minimum critical path delay, and lower power optimization.

As the GCM module takes advantage of the AES encryption in different parts, a future research on combination of fault detection in the AES part and the GCM loop can be very useful in order to reduce the area overhead and delay of overall fault detection unit. The future research is to extend the concept of single bit fault detection which is proposed in Chapter 3 to a multiple-bit fault

detection scheme which is independent of the multiplier type used in the GCM loop.

We need to investigate the concept of fault detection for other types of multipliers discussed in Section 2.4. In this approach multiple bit parities are added to the GCM loop and the propagation of faults with respect to the parity bits are investigated to make the fault detection technique suitable for high performance and low complexity applications.

$$
\begin{array}{c}
\\
R1 \\
R2 \\
R3 \\
\cdots \\
R121 \\
R122 \\
\cdots \\
R127 \\
R128
\end{array}
\begin{bmatrix}
C1 & & & & & & \cdots & C122 & & & & \cdots & C128 \\
0 & h_{127} & h_{126} & h_{125} & h_{124} & \cdots & h_7 & h_6{+}h_{127} & h_5{+}h_{126} & \cdots & h_1{+}h_{122}{+}h_{127} \\
0 & h_{127} & h_{126}{+}h_{127} & h_{125}{+}h_{126} & h_{124}{+}h_{125} & \cdots & h_7{+}h_8 & h_6{+}h_7{+}h_{127} & h_5{+}h_6{+}h_{126}{+}h_{127} & \cdots & h_1{+}h_2{+}h_{122}{+}h_{123}{+}h_{127} \\
0 & h_{127} & h_{125}{+}h_{126}{+}h_{127} & h_{124}{+}h_{125}{+}h_{126} & h_{123}{+}h_{124}{+}h_{125} & \cdots & h_7{+}h_8{+}h_9 & h_6{+}h_7{+}h_8{+}h_{127} & h_5{+}h_6{+}h_7{+}h_{126}{+}h_{127} & \cdots & h_1{+}h_2{+}h_3{+}h_{122}{+}h_{123}{+}h_{124}{+}h_{127} \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
0 & 0 & 0 & 0 & 0 & \cdots & h_{127} & h_{126} & h_{125} & \cdots & h_{121} \\
0 & 0 & 0 & 0 & 0 & \cdots & h_{127} & h_{126}{+}h_{127} & h_{125}{+}h_{126} & \cdots & h_{121}{+}h_{122} \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & h_{125}{+}h_{126}{+}h_{127} \\
0 & 0 & 0 & 0 & 0 & \cdots & h_{127} & h_{126} & h_{125} & \cdots & h1{+}h_{126}{+}h_{127}
\end{bmatrix}
$$

Figure 5.1: Demonstation of Matrix $\mathbf{Q}^T\mathbf{U}$.

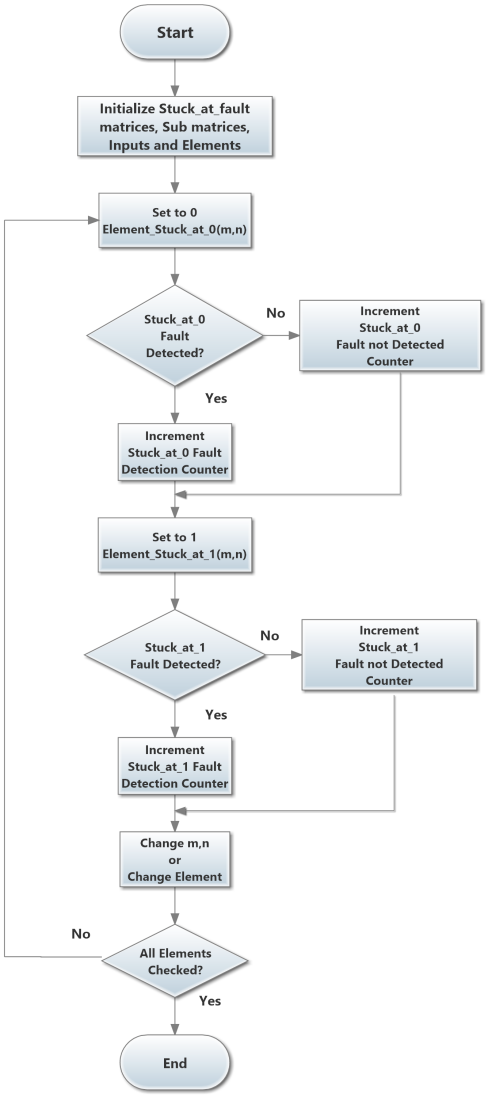

Figure 5.2: The flow chart of fault injection.

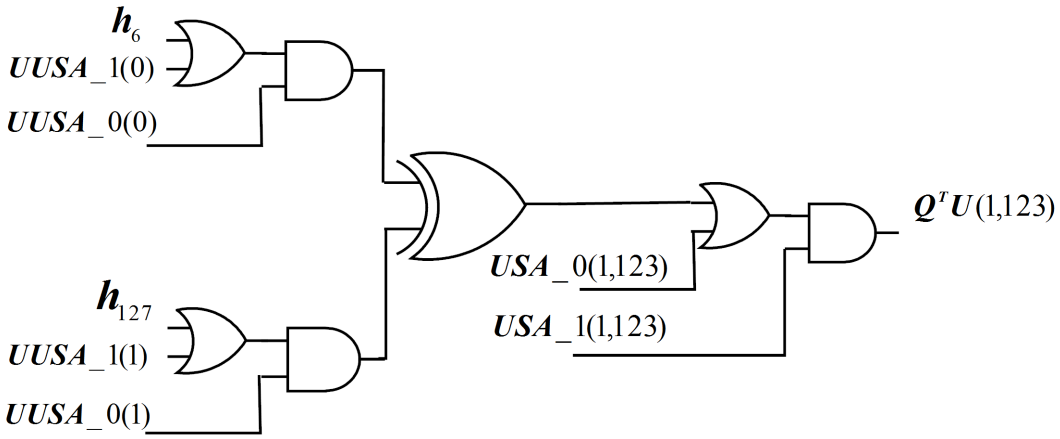

Figure 5.3: Gate level fault injection for matrix element $\mathbf{Q}^T\mathbf{U}(1, 123)$.

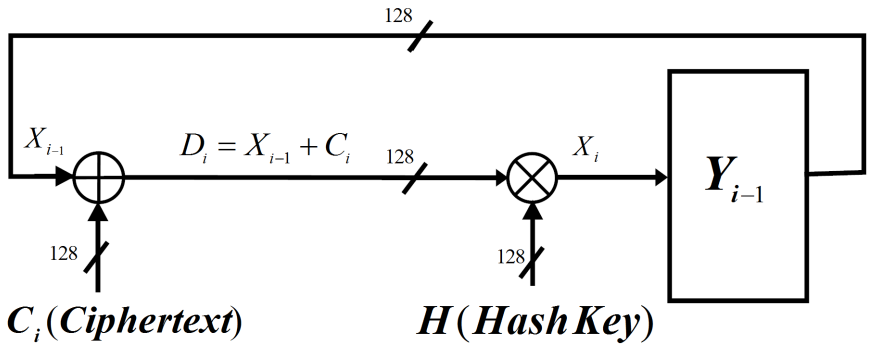

Figure 5.4: The GCM loop and related components

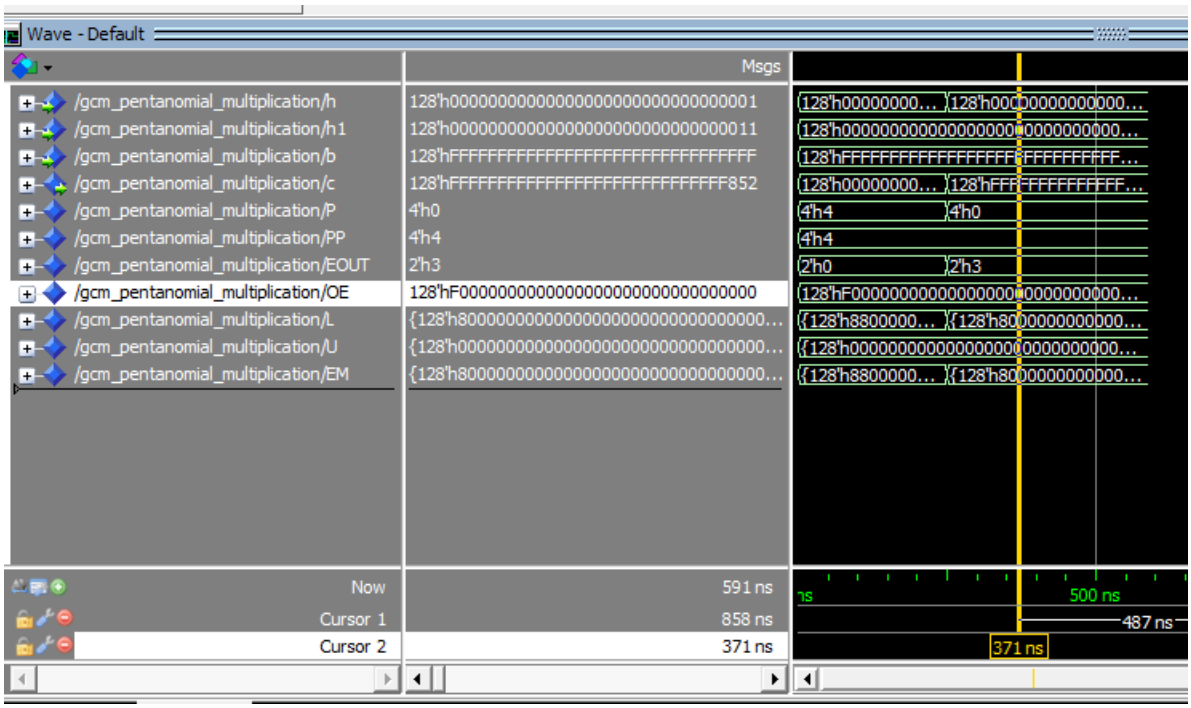

Figure 5.5: The simulation results.

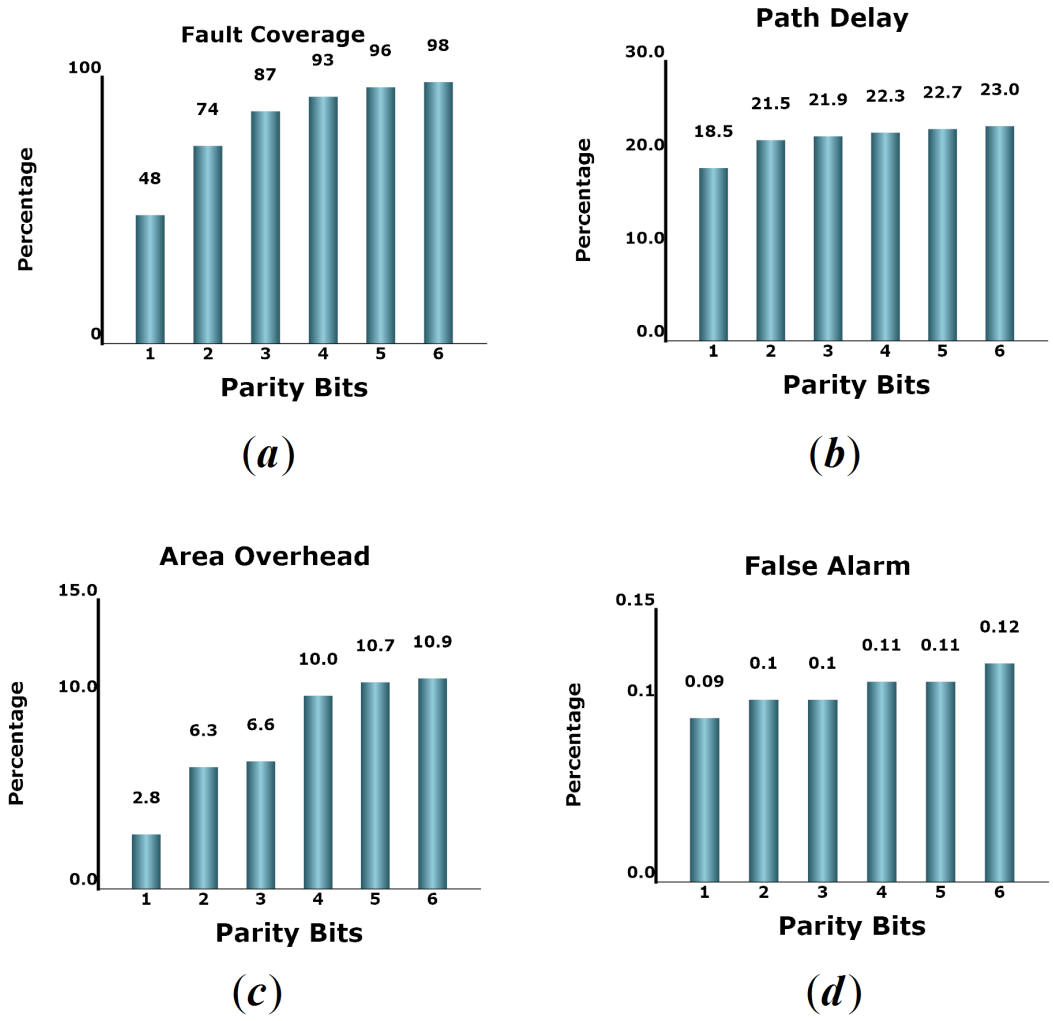

Figure 5.6: The simulation results graph: (a) fault coverage, (b) critical path delay, (c) area overhead, and (d) false alarm versus the number of parity bits.

Bibliography

[1] NIST Computer Security Division's (CSD) Security Technology Group (STG) (2013). "Proposed modes. Cryptographic Toolkit. NIST., 2013.

[2] Welch Brent B. and Jones Ken Chr. "Practical programming in Tcl and Tk 1 (4th ed.)". *Prentice Hall PTR. p. 291. ISBN 0-13-038560-3.*, 2003.

[3] Siavash Bayat-Sarmadi and M. Anwar Hasan. "On Concurrent Detection of Errors in Polynomial Basis Multiplication". *Very Large Scale Integration (VLSI) Systems, IEEE Transactions, pp. 413-426, vol.15*, 2007.

[4] S. Bellovin and R. Housley. "Guidelines for Cryptographic Key Management". *BCP 107, RFC 4107*, 2005.

[5] G. Bertoni, L. Breveglieri, I. Koren, P. Maistri, and V. Piuri. "Error analysis and detection procedures for a hardware implementation of the advanced encryption standard". *IEEE Transcations on Computers, pp. 492-505, vol. 52, no. 4*, 2003.

[6] E. Biham and A Shamir. "Differential Cryptanalysis of the Data Encryption Standard - Advances in Cryptology - CRYPTO '92". *12th Annual International Cryptology Conference, Santa Barbara, California, USA, Proceedings. pp. 487-496*, 1992.

[7] M. Poolakkaparambil C. T. Veedon and J. Mathew A.M. Jabir. "On the design of Trojan tolerant finite field multipliers". *Computing, Communication, Control and Compressed Sensing (iMac4s), 2013 International Multi-Conference , pp. 450-454*, 2013.

[8] S. Castagnoli, G. ; Braeuer and M. Herrman. "Optimization of Cyclic Redundancy-Check Codes with 24 and 32 parity bits". *IEEE Trans. on Communications, pp. 88-92, vol. 41, no. 6*, 1993.

[9] Jim-Min Lin Chiou-Hiou-Yng Lee, Che Wun Chiou. "Concurrent Error Detection in a Polynomial Basis Multiplier over $GF(2^m)$". *JOURNAL OF ELECTRONIC TESTING: Theory and Applications, pp. 143-150, vol. 22*, 2006.

[10] ALTERA Corporation. *URL: http://www.altera.com/products/software/products/quartus2/qts-index.html*.

[11] Joan Daemen and Vincent Rijiman. "AES Proposal: Rijndael". *National Institute of Standards and Technology, pp. 1-26*, 2013.

[12] Lombadrdi Fabrizio and Muzio Jon C. "Concurrent error detection and fault location in an FFT architecture". *IEEE Journal , pp. 728-736, vol. 27, Issue: 5,*, 1992.

[13] FIPS Federal Information Processing. *URL: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.*

[14] Specication for the Advanced Encryption Standard (AES). Technical Report FIPSPUB197, 2001.

[15] Christophe Giraud and Hugues Thiebeauld. "Basics of Fault Attacks". *Oberthur Card Systems and Thales Microelectronics*, 2004.

[16] Mohsen Machhout Hassan Mestiri, Noura Benhadjyousef and Rached Tourki. "A Robust Fault Detection Scheme for the Advanced Encryption Standard". *International Journal of Computer Network and Information Security, pp. 49-55*, 2013.

[17] M. Karpovsky, K. J. Kulikowski, and A. Taubin. "Differential Fault Analysis Attack Resistant Architectures for the Advanced Encryption Standard". *CARDIS 04: Sixth smart Card Research and Advanced Application IFIP Conference, Toulouse, France, pp. 177-192, vol. 153*, 2004.

[18] Philip Koopman and Tridib Chakravarty. "Cyclic Redundancy Code (CRC) Polynomial Selection For Embedded Networks". *The International Conference on Dependable Systems and Networks*, 2004.

[19] R. Lidl and H. Niederreiter. "Introduction to Finite Fields and Their Applications". *Cambridge University Press*, 1994.

[20] S. Lin and D. J. Costello. "Error Control Coding, Prentice Hall, second edition, Upper Saddle River, NJ, USA.

[21] D. Taylor M. Gossel, S. Fenn. "On-line Error Detection for Finite Field Multipliers". *Defect and Fault Tolerance in VLSI Systems. Proceedings of IEEE International Symposium, pp. 307-311*, 1997.

[22] D. McGrew and J. Viega. "The Galois/Counter Mode of Operation (GCM), National Institude of Standard and Technology". *URL: http://www.csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-revised-spec.pdf*, 2005.

[23] D. A. McGrew and J. Viega. "The Security and Performance of the Galois/Counter Mode(GCM) of Operation (Full Version)". *URL: http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/gcmgcm-ad.pdf*, 2008.

[24] H. Mestiri, N. Benhadjyoussef, M. Machhout, and R. Tourki. "An FPGA implementation of the AES with fault detection countermeasure". *IEEE International Conference on Control, Decision and Information Technologies (CoDIT), pp. 264-270*, 2013.

[25] Mehran Mozaffari-Kermani and Arash Reyhani-Masoleh. "Efficient and High-Performance Parallel Hardware Architectures for the AES-GCM". *IEEE Transactions. on Computers, pp. 1089-1103, vol. 55, no. 8*, 2006.

[26] Mehran Mozaffari-Kermani and Arash Reyhani-Masoleh. "A Lightweight High-Performance Fault Detection Scheme for the Advanced Encryption Standard Using Composite Fields". *IEEE Transaction on Computers, pp. 85-91, vol. 9*, 2009.

[27] C. Negre N. Meloni and M. A. Hassan. "High Performance GHASH Function for Long Messages". *In Proc. of ACNS 2010, pp. 154-167*, 2010.

[28] J. H. Patel and L.Y. Fung. "Concurrent Error Detection in ALUs by Recomputing with Shifted Operands". *IEEE Trans. Computers, pp. 589-595, vol. C-31, no. 7*, 1982.

[29] S. A. Reddy and M. A. Kumar. Efficient fault detection scheme for reliable AES architecture". *IEEE International Conference on Emerging Trends in Electrical and Computer Technology (ICETECT), pp. 1004-1009*, 2011.

[30] A. Reyhani-Masoleh and M. A Hasan. "Fault Detection Architectures for Field Multiplication Using Polynomial Bases". *IEEE Trans. on Computers, pp. 1089-1103, vol. 55, no.9*, 2006.

[31] Arash Reyhani-Masoleh and M.Anwar Hasan. "Low Complexity Bit Parallel Architectures for Polynomial Bases Multiplication over $GF(2^m)$". *IEEE Trans. on Computers, pp. 945-958, vol. 53, no.8*, 2004.

[32] Terry Ritter. "The Great CRC Mystery". *Dr. Dobb's Journal 11 (2): pp. 26-34, pp, 76-83*, 2009.

[33] NIST AES Fact Sheet. *URL: http://csrc.nist.gov/CryptoToolkit/aes/aesfact.html*.

[34] Iqbal Muhammad Umair. "On Software Implementation of High Performance GHASH Algorithms". *University of Waterloo, Electronic Thesis Dessertations*, 2012.

[35] NIST Special Publication 800-38A, Version 1. "Recommendation for Block Cipher Modes of Operation Methods and Technique, 2001.

[36] W. T. Vetterling W. H. Press, S. A. Teukolsky and BP. Flannery. *The Art of Scientific Computing (3rd ed.), New York: Cambridge University Press. ISBN 978-0-521-88068-8.*

[37] Sheng Wang. "An Architecture for the AES-GCM Security Standard". *University of Waterloo, Electronic Thesis Dessertations*, 2006.

[38] H. Wu. "Bit-Parallel Finite Field Multiplier and Squarer Using Polynomial Basis". *IEEE Trans. Computers, pp. 750-758, vol. 51, no. 7*, 2002.

[39] Chih-Hsu Yen and Bing-Fei Wu. "Simple error detection methods for hardware implementation of Advanced Encryption Standard". *IEEE Transcations on Computers, pp. 720-731, vol. 55, no. 6*, 2006.

Appendix A

VHDL Implementation

```
--================================================================
-- Multiplier, Pentanomial (GCM_pentanomial_multiplier.vhd)
--
-- Computes the multiplication in GF(2^128) for GCM Loop
--
-- The irreducible polynomial : f(x)= x^128 + x^7 + x^2 + x + 1
--================================================================
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;



entity GCM_pentanomial_multiplication is
generic (M: natural := 128);

--"h" stands for Hask Key "H" of the GCM module
port (
 h1,h,b: in std_logic_vector(M-1 downto 0);
  c: out std_logic_vector(M-1 downto 0);
 signal P: out    std_logic_vector (3 downto 0);

-- Output Parity Prediction:PP(0) to PP(2);
  signal PP: out   std_logic_vector (3 downto 0)
  );

end GCM_pentanomial_multiplication;

architecture structure of GCM_pentanomial_multiplication is

type Umatrix is array (M downto 1,M downto 1) of std_logic;
type Lmatrix is array (M downto 1,M downto 1) of std_logic;
type USAmatrix is array (M downto 1,M downto 1) of std_logic;

  signal D:     std_logic_vector (M-1 downto 0);
  signal E:     std_logic_vector (M-2 downto 0);

-- Output Parity :P(0) to P(2);
```

```
 signal pp_temp : std_logic_vector(3 downto 0);
signal  p_temp : std_logic_vector(3 downto 0);

--  Copy of Multiplier output for parity calculation purpose;
  signal CO:    std_logic_vector (M-1 downto 0)
                        := x"00000000000000000000000000000000";
--  The OE matrix for parity prediction purpose
  signal OE:    std_logic_vector (M-1 downto 0)
                        := x"00000000000000000000000000000000";
  signal EOUT:  std_logic_vector (1 downto 0)  :="00";

-- CRC Pattern for f(x)=x^2+x+1 : 2 bit parity construct
constant CRC1:  std_logic_vector(M-1 downto 0)
                        := x"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF";
constant CRC2:  std_logic_vector(M-1 downto 0)
                        := x"B6DB6DB6DB6DB6DB6DB6DB6DB6DB6DB6";
constant CRC3:  std_logic_vector(M-1 downto 0)
                        := x"55555555555555555555555555555555";

-- CRC pattern for f(x)=x^3+x+1  : 3 bit parity construct
--constant CRC1:  std_logic_vector(M-1 downto 0)
                        := x"74E9D3A74E9D3A74E9D3A74E9D3A74E9";
-- constant CRC2:  std_logic_vector(M-1 downto 0)
                        := x"3A74E9D3A74E9D3A74E9D3A74E9D3A74";
--constant CRC3:  std_logic_vector(M-1 downto 0)
                        :=  x"BAEBAEBAEBAEBAEBAEBAEBAEBAEBAEBA";
signal U : USAmatrix;
signal USA_0: USAmatrix;
signal USA_1: USAmatrix;
signal LSA_0 :USAmatrix;
signal LSA_1 :USAmatrix;
signal EMSA_0 :USAmatrix;
signal EMSA_1  :USAmatrix;
signal U_temp :USAmatrix;
signal L ,EM: Lmatrix;


begin
-- Q 'Transpose' multiply by U
Initialization_of_QTU_matrix: process(U,USA_0,USA_1,LSA_0,LSA_1,U_temp)

begin



for j in 1 to 128 loop
for i in 1 to 128 loop
U(j,i)<='0';
end loop;
end loop;
---1 st Row
U(1,1)<='0';
for i in 2 to 122 loop
U(1,i)<=h(127-(i-2));
```

```
end loop ;
for i in 123 to 127 loop
U(1,i)<= h(127-(i-2))xor h(127-(i-123));
end loop;
U(1,128)<= h(1)xor h(122) xor h(127);

---2nd Row
U(2,1)<='0';U(2,2)<=h(127);
for i in 3 to 122 loop
U(2,i)<=h(127-(i-2))xor h(127-(i-3));
end loop ;
U(2,123)<=h(6)xor h(7)xor h(127);
for i in 124 to 127 loop
U(2,i)<=h(127-(i-2))xor h(127-(i-3))xor h(127-(i-124)) xor h(127-(i-123));
end loop;
U(2,128)<=h(1) xor h(2) xor h(122) xor h(123) xor h(127);

---3rd Row
U(3,1)<='0';U(3,2)<=h(127);U(3,3)<=h(127) xor h(126);
for i in 4 to 122 loop
U(3,i)<=h(127-(i-2))xor h(127-(i-3))xor h(127-(i-4));
end loop ;
U(3,123)<=h(6)xor h(7)xor h(8)xor h(127);
U(3,124)<=h(5) xor h(6)xor h(7)xor h(127) xor h(126);

for i in 125 to 127 loop
U(3,i)<=h(127-(i-2))xor h(127-(i-3))xor h(127-(i-4))xor(
 h(127-(i-125))xor h(127-(i-124)) xor h(127-(i-123)));
end loop;
U(3,128)<=h(1) xor h(2) xor h(3) xor h(122) xor h(123)xor h(124) xor h(127);

---4th Row
U(4,1)<='0';U(4,2)<='0';U(4,3)<=h(127);U(4,4)<=h(127) xor h(126);
for i in 5 to 123 loop
U(4,i)<=h(127-(i-3))xor h(127-(i-4))xor h(127-(i-5));
end loop ;

U(4,124)<=h(6)xor h(7)xor h(8) xor h(127);
U(4,125)<=h(5)xor h(6)xor h(7) xor h(126) xor h(127);
for i in 126 to 128 loop
U(4,i)<=h(127-(i-3))xor h(127-(i-4))xor h(127-(i-5))xor(
 h(127-(i-126))xor h(127-(i-125)) xor h(127-(i-124)));
end loop;

---5th Row
U(5,1)<='0';U(5,2)<='0';U(5,3)<='0';U(5,4)<=h(127);
for i in 5 to 124 loop
U(5,i)<=h(127-(i-5))xor h(127-(i-4));
end loop ;
U(5,125)<= h(6)xor h(7) xor h(127);
U(5,126)<=h(5)xor h(6)xor h(126) xor h(127);
U(5,127)<= h(4)xor h(5) xor h1(125)xor h(126)xor h(127);
U(5,128)<=h(3) xor h(4) xor h(124)xor h(125) xor h(126);
```

```
---6th Row
U(6,1)<='0';U(6,2)<='0';U(6,3)<='0';U(6,4)<='0';
for i in 5 to 125 loop
U(6,i)<=h(127-(i-5));
end loop ;
U(6,126)<= h(6)xor  h(127);
U(6,127)<=h(5)xor h(126) xor h(127);
U(6,128)<=h(4)xor h(125)xor h(126) xor h(127);

---7th Row

for i in 1 to 126 loop
U(7,i)<='0';
end loop ;
U(7,127)<= h(127);
U(7,128)<=h(126) xor h(127);

---8th Row

U(8,1)<='0';
for i in 2 to 122 loop
U(8,i)<=h(127-(i-2));
end loop ;
for i in 123 to 128 loop
U(8,i)<=h(127-(i-2))xor h(127-(i-123));
end loop ;

--9th Row
U(9,1)<='0';U(9,2)<='0';
for i in 3 to 123 loop
U(9,i)<=h(127-(i-3));
end loop ;
for i in 124 to 128 loop
U(9,i)<=h(127-(i-3))xor h(127-(i-124));
end loop ;

--10th Row
U(10,1)<='0';U(10,2)<='0';U(10,3)<='0';
for i in 4 to 124 loop
U(10,i)<=h(127-(i-4));
end loop ;
for i in 125 to 128 loop
U(10,i)<=h(127-(i-4))xor h(127-(i-125));
end loop ;

--11th Row

for i in 1 to 4 loop
U(11,i)<='0';
end loop;
for i in 5 to 125 loop
U(11,i)<= h(127-(i-5));
end loop ;
```

```
for i in 126 to 128 loop
U(11,i)<=h(127-(i-5))xor h(127-(i-126));
end loop ;
---12th Row

for i in 1 to 126 loop
U(12,i)<='0';
end loop ;
U(12,127)<= h(127);
U(12,128)<=h(126);

---13th Row

for i in 1 to 127 loop
U(13,i)<='0';
end loop ;
U(13,128)<= h(127);

---14th  to 120th Row
for j in 14 to 120 loop
for i in 1 to 128 loop
U(j,i)<='0';
end loop;
end loop;

---121st Row

for i in 1 to 121 loop
U(121,i)<='0';
end loop ;

for i in 122 to 128 loop
U(121,i)<= h(127-(i-122));
end loop ;

---122nd Row

for i in 1 to 121 loop
U(122,i)<='0';
end loop ;
U(122,122)<= h(127);
for i in 123 to 128 loop
U(122,i)<=h(127-(i-122))xor h(127-(i-123));
end loop ;

---123rd Row

for i in 1 to 121 loop
                U(123,i)<='0';
        end loop ;
U(123,122)<=h(127);
U(123,123)<= h(127)xor h(126);

for i in 124 to 128 loop
```

```
U(123,i)<=h(127-(i-122))xor h(127-(i-123))xor h(127-(i-124));
end loop ;

---124th Row
for i in 1 to 122 loop
U(124,i)<='0';
end loop ;
U(124,123)<= h(127);
U(124,124)<= h(127)xor h(126);

for i in 125 to 128 loop
U(124,i)<=h(127-(i-123))xor h(127-(i-124))xor h(127-(i-125));
end loop ;


---125th Row

for i in 1 to 123 loop
U(125,i)<='0';
end loop ;
U(125,124)<= h(127);
U(125,125)<= h(127) xor h(126);

for i in 126 to 128 loop
U(125,i)<= h(127-(i-124))xor h(127-(i-125)) xor h(127-(i-126));
end loop ;

---126th Row

for i in 1 to 124 loop
U(126,i)<='0';
end loop ;
U(126,125)<= h(127);
U(126,126)<= h(127) xor h(126);
U(126,127)<= h(127)xor h(126)xor h(125);
U(126,128)<= h(124)xor h(125)xor h(126);


--127th Row

for i in 1 to 125 loop
U(127,i)<='0';
end loop ;
U(127,126)<= h(127);
U(127,127)<= h(127)xor h(126);
U(127,128)<= h(125)xor h(126)xor h(127);

---128th Row

for i in 1 to 121 loop
U(128,i)<='0';
end loop ;

U(128,122)<= h(127);
```

```
for i in 123 to 127 loop
U(128,i)<= h(127-(i-122));
end loop ;
U(128,127)<= h(122)xor h(127);
--U(128,127)<=U(128,127)xor h(127);

U(128,128) <=h(121)xor h(126)xor h(127);



end process Initialization_of_QTU_matrix;


Initialization_of_L_matrix:process(L,h)
variable t,s: integer ;

begin

t:=1;
for j in 1 to 128 loop
for i in 1 to  t loop
L(j,i)<= h(t-i);
end loop;
s:=t+1;
for i in s to 128 loop
L(j,i)<='0';
end loop;
t:=t+1;
end loop;


end process Initialization_of_L_matrix;


Initialization_of_E_matrix: process(L,U)
begin
for j in 1 to 128 loop
 for i in 1 to 128 loop
  EM(j,i)<= L(j,i) xor U(j,i);
 end loop;
end loop;


end process Initialization_of_E_matrix;

Multiplication_output:process(EM,b)
 variable c_temp : std_logic_vector(M-1 downto 0);
variable cc_temp: std_logic;
begin
cc_temp :='0';
for j in 1 to 128 loop
for i in 1 to 128 loop
cc_temp:= (EM(j,i) and b(i-1))xor cc_temp;
end loop;
```

```
c(j-1)<= cc_temp;
c_temp(j-1):=cc_temp;
cc_temp:='0';
end loop;
CO <= c_temp;
end process Multiplication_output;



Parity: process(CO)

variable pt : std_logic_vector(3 downto 0);
variable crc_temp1 : std_logic_vector(M-1 downto 0);
variable crc_temp2 : std_logic_vector(M-1 downto 0);
variable crc_temp3 : std_logic_vector(M-1 downto 0);
variable k :integer;
 begin
  for i in 0 to 3 loop
   pt(i):='0';
   end loop ;
    for i in 0 to M-1 loop    -- Parity of Multiplier output
     pt(0):= CO(i) xor pt(0);


      end loop;
  -- Parity of Multiplier output with CRC pattern # 1 applied.
       crc_temp1:= CO  and CRC1;

     for i in 0 to M-1 loop

     pt(1):= crc_temp1(i) xor pt(1);
      end loop;

     pt(2):='0';
-- Parity of Multiplier output with CRC2 pattern # 2 applied.

 crc_temp2:= CO  and CRC2;

     for i in 0 to M-1 loop

     pt(2):= crc_temp2(i) xor pt(2);
      end loop;


-- Parity of Multiplier output with CRC pattern # 3 applied.
   pt(3):='0';
      crc_temp3:= CO and CRC3;
     for i in 0 to M-1 loop

     pt(3):= crc_temp3(i) xor pt(3);
      end loop;
--pt(2):='0';-- should be removed for CRC checking
--pt(3):='0';
      P <= pt;
```

```
                p_temp <=pt;
 end process Parity;


Parity_Prediction: process(h,b)
variable pH : std_logic_vector(1 downto 0):="00";
variable pt : std_logic_vector(3 downto 0);
variable crc_temp1 : std_logic_vector(M-1 downto 0);
variable crc_temp2 : std_logic_vector(M-1 downto 0);
variable crc_temp3 : std_logic_vector(M-1 downto 0);
variable OEB :       std_logic_vector(M-1 downto 0);
variable OEB_temp :       std_logic_vector(M-1 downto 0);
variable OEB_temp1 :       std_logic_vector(M-1 downto 0);
variable OEB_temp2 :       std_logic_vector(M-1 downto 0);
variable OEB_temp3 :       std_logic_vector(M-1 downto 0);
variable OEB_temp4 :       std_logic_vector(M-1 downto 0);

variable k,k1,s1,k2,s2: integer;




   begin

-- initialization of parity variable
  for i in 0 to 3 loop
   pt(i):='0';
   end loop ;

  for i in 0 to 1 loop
   pH(i):='0';
  end loop ;


    --pH:="00";   -- OE construction
    OEB_temp := h1;
    for i in 0 to M-1 loop

-- Calculating the Parity of H :pH(0)
     pH(0):= OEB_temp(i) xor pH(0);
   end loop;
     OEB_temp1:=h1;
     OEB_temp1(0):=pH(0);

   for i in 1 to 121 loop
   OEB_temp1(i):=OEB_temp1(i-1) xor h1(M-i);
   end loop;
 OEB_temp1(122):= h1(0) xor h1(1) xor h1(2) xor h1(3)
                        xor h1(4) xor h1(5) xor h1(127);
 OEB_temp1(123):= h1(0) xor h1(1) xor h1(2) xor h1(3)
                        xor h1(4) xor h1(126) xor h1(127);
 OEB_temp1(124):= h1(0) xor h1(1) xor h1(2) xor h1(3)
                        xor h1(125) xor h1(126) xor h1(127);
 OEB_temp1(125):= h1(0) xor h1(1) xor h1(2) xor h1(124)
```

```
                        xor h1(125) xor h1(126) xor h1(127);
 OEB_temp1(126):= h1(0) xor h1(1) xor h1(123) xor h1(124)
                        xor h1(125) xor h1(126) xor h1(127);
 OEB_temp1(127):= h1(0) xor h1(122) xor h1(123) xor h1(124)
                        xor h1(125) xor h1(126) ;
OE <= OEB_temp1;
 -- End of OE Construction


-- bitwise AND of Register OE & input b
   OEB_temp2:= OEB_temp1 and b;

-- Prediction Parity of Multiplier output
    for i in 0 to M-1 loop
      pt(0):= OEB_temp2(i) xor pt(0);
     end loop;
-- Prediction Parity of Multiplier output with1 CRC pattern # 1 applied.
     crc_temp1:= OEB_temp2 and CRC1;
     for i in 0 to M-1 loop
      pt(1):= crc_temp1(i) xor pt(1);
     end loop;

OEB_temp3 := OEB_temp1;
--Added

for i in 0 to 42 loop
OEB_temp3(0) := OEB_temp3(0) xor h1(i*3);
end loop;

------
OEB_temp3(1):= OEB_temp1(1) xor h1(127);

--Added
for i in 1 to 42 loop
OEB_temp3(1) := OEB_temp3(1) xor h1(i*3-1);
end loop;
------

OEB_temp3(2):= OEB_temp1(2) xor h1(127) xor h1(126);

--Added
for i in 1 to 42 loop
OEB_temp3(2) := OEB_temp3(2) xor h1(i*3-2);
end loop;
------


OEB_temp3(3):= OEB_temp1(3) xor h1(126) xor h1(125);--changed from (127)
--Added
for i in 1 to 42 loop
OEB_temp3(3) := OEB_temp3(3) xor h1(i*3-3);
end loop;
------
```

```
k:=1;
for j in 4 to 120 loop
OEB_temp3(j) := OEB_temp1(j)xor h1(127-(j-1))xor h1(127-(j-2))xor h1(127-(j-4)) ;
--Added
for i in 1+k to 42 loop
OEB_temp3(j) := OEB_temp3(j) xor h1(i*3-j);
end loop;
k:=j/3;
end loop;
------


 OEB_temp3(121):= h1(127) xor h1(7)xor h1(8)xor h1(10)xor h1(1)
                          xor h1(3)xor h1(4)xor h1(6)xor h1(0);-- h2,h5
 OEB_temp3(122):= h1(126) xor h1(127)xor h1(6)xor h1(7)xor h1(9)
                          xor h1(0)xor h1(2)xor h1(3)xor h1(5);--h1,h4
 OEB_temp3(123):= h1(125) xor h1(126) xor h1(127)xor h1(5)xor h1(6)
                          xor h1(8)xor h1(1)xor h1(2)xor h1(4);--h0,h3
 OEB_temp3(124):= h1(124) xor h1(125)xor h1(126) xor h1(4)xor h1(5)
                          xor h1(7)xor h1(0)xor h1(1)xor h1(3); --h2
 OEB_temp3(125):= h1(123)xor h1(124) xor h1(125)xor h1(127)xor h1(3)
                          xor h1(4)xor h1(6)xor h1(0)xor h1(2); --h1
 OEB_temp3(126):= h1(122) xor h1(123)xor h1(124) xor h1(126) xor h1(127)
                          xor h1(2)xor h1(3)xor h1(5)xor h1(1);--h0
 OEB_temp3(127):= h1(121) xor h1(122) xor h1(123) xor h1(125)xor h1(126)
                          xor h1(1)xor h1(2)xor h1(4)xor h1(0);


-- Prediction Parity of Multiplier output with CRC pattern # 2 applied.
     crc_temp2:= OEB_temp3 and b;
     for i in 0 to M-1 loop
     pt(2):= crc_temp2(i) xor pt(2);
      end loop;

-- Prediction Parity of Multiplier output with CRC pattern # 3 applied.

for i in 0 to 127 loop
OEB_temp4(i) := '0';
end loop;
--Added
k1:=0;
s1:=0;
for j in 0 to 120 loop
for i in 0 to 63-s1 loop
OEB_temp4(2*j) := OEB_temp4(2*j) xor h1(i*2);
end loop;
s1:=s1+1;
end loop;

------

s2:=0;
for j in 1 to 120 loop
for i in 0 to 62-s2 loop
```

```
OEB_temp4(2*j-1) := OEB_temp4(2*j-1) xor h1(i*2+1);

end loop;

s2:=s2+1;
end loop;




 OEB_temp4(122):= h1(127) xor h1(0)xor h1(2)xor h1(4);
 OEB_temp4(123):= h1(126) xor h1(1)xor h1(3);
 OEB_temp4(124):= h1(125)xor h1(127)xor  h1(0)xor h1(2);
 OEB_temp4(125):= h1(124) xor h1(126) xor h1(1);
 OEB_temp4(126):= h1(123)xor h1(125)xor h1(127)xor h1(0);
 OEB_temp4(127):= h1(122) xor h1(124) xor h1(126) xor h1(127);



pt(3):='0';

 crc_temp3:= OEB_temp4 and b;
     for i in 0 to M-1 loop
     pt(3):= crc_temp3(i) xor pt(3);
      end loop;
      PP <= pt;
                pp_temp <=pt;
 end process Parity_Prediction;

Error: process(pp_temp)
variable E_state : std_logic_vector(1 downto 0);

begin

if ( pp_temp   =  p_temp   )
then
E_state := E_state and "00";
EOUT <= E_state ;
else
  E_state := E_state or  "11";
 EOUT <= E_state;

end if;
end process Error;

end structure;
```

Appendix B

Fault Injection TLC

```
# set the simulation step size as a global variable
# step is used by both runSim and verifySim listed below
set step 10
set Error_Detected 0
set j 0
set runtime 5
   # a count of the elapsed runtime
   set runtime 0
   # import the global variable step
   global step
#set array m(128)

#set m1  0x00000000000000005670000000000001
# set m1  0x76543210
# set m2  0xFECBA987
  set m1 0x0000001001100000000FECBA9876543210
#0x0000001000000000000FECBA9876543210


set m   0x00000000000000000000000010000000
   restart -force -nowave
   add wave -logic h
   add wave -logic h1
   add wave -logic b
   add wave -logic c
   add wave -logic P
   add wave -logic PP
   add wave -logic EOUT

#m($0)=1
 set filename "Multiplier_Error.txt"
        # open the filename for writing
         set fileId [open $filename "w"]

   # apply all 64 (2^6) possible inputs to the design
   for {set i 0} {$i < 1000} {incr i} {
     # force signals based on a mask of the integer i
      #force -freeze h [expr $m1  || $m  ]
```

```
       #${m2}= ${m1} & ${m}
#force -freeze h [ format "%0x" [expr {$m >> $i | $m1}]]
      #main force -freeze h [ expr $m >> 1 | $m1]
incr j
if { $j < 390} {

         force -freeze h [ expr $m >> $i | $m1]
     }

 if { $j >= 390 }  {
set m1  0x00000000000000005670000000000001

force -freeze h  [expr  $m1]
}
#force -freeze h  [format "%0x" [expr $m1]]+[format "%0x" [expr $m2]]



   #  force -freeze h $($m >> $i | $m1 )



     force -freeze h1 128'h00000000000000005670000000000001
     force -freeze b  128'h00004400000020000009000000000012


   run 250




set simOut [examine  -binary /EOUT]


if { $simOut != "2'b00"} {

       incr Error_Detected
      # puts -nonewline $fileId $Error_Detected
      #   puts -nonewline $fileId $simOut

     }

}


        # Create Error lof file when performing exhaustive test
      #   set filename "Multiplier_Error.txt"
        # open the filename for writing
     #    set fileId [open $filename "w"]
        # write the number of error occurance " the data to the file -
        puts -nonewline $fileId $Error_Detected
     #    puts -nonewline $fileId $simOut

        # close the file, ensuring the data is written out before you continue
```

```
    #  with processing.
    close $fileId

# after the simulation is complete, view the results
view wave
```