The University Of Sheffield.

A Study of Polynomial Residue Number Systems over Binary Galois Fields $GF(2^m)$ for Cryptography

By
Junfeng Chu

A Thesis submitted to the University of Sheffield in partial fulfilment of the requirements for the degree of Doctor of Philosophy.

January 2012

Abstract

This thesis is concerned with GF(2^m) Polynomial Residue Number Systems (PRNS) and their application in cryptography to provide resistance against side-channel-analysis and protection against fault attacks.

PRNS operations over GF(2^m) required in a number of cryptography primitives are investigated. A partial-conversion method is introduced to simplify the costly conversion operation and this is then combined with a partial modular reduction technique and applied to design and implement a PRNS based GF(2^m) multiplier with improved performance.

The Advanced Encryption Standard (AES) is used as vehicle to analyse and quantify the PRNS overhead where different AES architectures are proposed and implemented. The PRNS based AES is shown to achieve excellent multiple error coverage with a reasonable overhead. It is also argued in the thesis, that PRNS AES designs provide an intrinsic resistance against probing attacks and, due to the introduction of redundant information and the residue representation replacing the original representation, exhibit increased confusion and hence enhanced design security.

Acknowledgements

I would like to thank my supervisor Dr. Mohammed Benaissa for his patience, guidance and constructive suggestions throughout the research period, for the freedom of research and frequent discussions that made the whole PhD journey such an enjoyable experience. Acknowledgement is also given to the University of Sheffield for all kinds of support.

I wish to express my gratitude to my colleagues: Nabil, Stas and Zia for their help and encouragement during my study. I also need to thank Neil Powell for his help while I was doing lab demonstrating.

To my lovely flat mates, I thank you all for putting up with my annoying living habits and sharing a wonderful life in Sheffield with me. A big thank you to all my friends I have come to know and rely on over these years. A special thanks is given to Xiaoshu, who helped me with my writing up and took care of me in the last stage of my PhD study.

To my best friend Chang Liu, who sadly passed away when I was studying in the UK.

Finally, a special sincere thanks goes to my parents for their enduring love, tolerance and support throughout my education. I wouldn't be here if it wasn't for you mum and dad, thanks. I am proud of you! I love you!

Sincerely

Junfeng Chu

1st Feb. 2012

Dedication

To my parents, Jianming Zhang and Chengdong Chu, whose unconditional love and unwavering support has allowed me to embark on this inspiring doctoral journey

To the memory of my best friend, Chang Liu.

To those who are loving me

To those who loved me

Table of Contents

Thesis Figure Captions

Thesis Table Captions

Thesis Algorithm Captions

Acronyms

AES	Advanced Encryption Standard
ASIP	Application Specific Instruction Processor
CFB	Cipher Feedback Mode
CRT	Chinese Remainder Theorem
CTR	Counter Mode
DSP	Digital Signal Processing
DUT	Design Under Test
ECC	Elliptic Curve Cryptography
FIPS	Federal Information Processing Standard
FLT	Fermat's Little Theorem
FPGA	Field Programmable Gate Array
GCD	Greatest Common Divisor
GF	Galois Field
LCM	Least Common Multiple
LSB	Least Significant Bit
LUT	Look Up Table
MRC	Mixed Radix Conversion
MRD	Mixed Radix Digit
MRS	Mixed Radix Number System
MSB	Most Significant Bit
MUX	Multiplexer
NB	Normal Basis
NIST	National Institute of Standards and Technology
OFB	Output Feedback Mode
PB	Polynomial Basis
PRNS	Polynomial Residue Number System
RESO	Re-computing with Shifted Operands
RNS	Residue Number System
RPRNS	Redundant Polynomial Residue Number System
RTL	Register Transfer Level
SoC	System on Chip
SRC	Single Radix Conversion
VHDL	Very High Speed Integrated Circuit Hardware Description Language

Chapter 1
Introduction

1.1 Introduction

Polynomial Residue Number System (PRNS) over binary fields[1] is a form of Residue Number System (RNS), where in each PRNS channel[2] the modular ring is generated by an irreducible polynomial rather than a primitive number as in normal RNS over integers. The Chinese Remainder Theorem (CRT), which is applicable in RNS, is applicable to PRNS as well [1]. RNS structures have the advantages of less power dissipation and less time consumption compared to traditional systems by using smaller operands and reducing the complexity of circuits [2]. Due to the nature of independence between RNS channels and scope for randomisation, RNS architectures have also been advocated for improving side-channel resistance in cryptosystems [3] and implementing fault tolerance in DSP and communication systems [4].

PRNS over binary field shares most of the attractive properties with the normal RNS over integers. The main aim of the research in this thesis is to explore and apply those attractive properties to the cryptography area to enhance the security level of cryptosystems by providing improved protection against fault and a number of side-channel attacks.

$GF(2^m)$ Multiplication is the most frequently used field operation in most

[1] $GF(2^m)$ is also known as binary field
[2] Each independent unit in PRNS or RNS is often referred to as a *channel*

cryptography primitives and as such the design of PRNS $GF(2^m)$ multiplier is crucial to the overall performance. The normal modular reduction operation in normal $GF(2^m)$, which requires weighted polynomial information of the intermediate product, needs the conversion from PRNS. This conversion from PRNS is particularly costly and, hence, it acts as an obstacle to the acceptance of PRNS. To overcome this and to simplify the conversion circuit, research was undertaken into trying different irreducible polynomial sets for the PRNS channels. Broadly speaking, field-generating polynomials are characterized as either being *generic* or *special*. In contrast to generic irreducible polynomials the modular reduction process for special irreducible polynomial is simplified, a typical example is using irreducible trinomials. The experimental results obtained show that using trinomials as the channel irreducible polynomials greatly reduces the complexity of the conversion, therefore improving the performance of the PRNS $GF(2^m)$ arithmetic and ultimately the whole cryptography primitive. To fully understand the performance of the PRNS $GF(2^m)$ multiplication, a thorough investigation is carried out including actual implementation results under a range of scenarios; for example the case where the proposed multiplier uses different PRNS channel arithmetic architectures: bit-serial and bit-parallel. Another comparison between different architectures for the PRNS channels is also provided: architectures where the channel operations are performed in serial by sharing one generic channel and architectures where individual channels are used to perform all the channel operations in parallel. A novel modular reduction method for the PRNS multiplier is introduced to further simplify the conversion circuit, namely the partial conversion method, where only part of the intermediate product is converted and the final result remains in the PRNS format.

To apply the devised PRNS to cryptography systems, the AES algorithm has been

implemented using a full PRNS architecture. Due to the flexibility of implementing the AES, different architectures are constructed to analyse the performance of the PRNS based AES: a very low area 8-bit data path AES and a normal 32-bit data path AES are implemented as reference designs to analyse the additional overheads. By adding a redundant channel, the proposed PRNS based AES is capable of detecting multiple errors with a reasonable overhead. The PRNS AES cores have an intrinsic resistance against probing attacks because all the transformations are performed independently in distributed PRNS channels. In addition, due to the introduction of the redundant channel and the residue representation replacing the original representation, more confusion is added to the system, which will also enhance the system's security level.

1.2 Main Research Contributions

The PRNS architecture brings a novel way of realising GF(2^m) circuits, it changes the architecture of cryptography schemes that uses GF(2^m) arithmetic from the bottom level. This research started by studying and implementing the basic GF(2^m) arithmetic using PRNS architecture; then the error detection and error correction capability provided by the PRNS architecture is explored; the PRNS architecture is then applied to the selected cryptography scheme which is the AES to provide the crypto-system with anti-fault-attack capability and other side channel analysis resistance. To the author's knowledge, the low area AES work that is proposed in Chapter 5 is known as the smallest FPGA AES design so far. The proposed PRNS architecture AES (Chapter 6) is shown to have excellent error detecting rate with promising overhead in hardware. The 8-bit PRNS AES design is known as the smallest AES scheme with multiple error detection capability over FPGA platforms.

The main contributions of the research that will be outlined in this thesis include:

- Design and implementation of a PRNS GF(2^m) multiplier using generic moduli sets, where m is 163, which is designed particularly for ECC (Elliptic Curve Cryptography) operations. Channel serial and channel parallel architecture of PRNS are implemented and the synthesis results are compared.
- Design and implementation of a PRNS GF(2^m) multiplier using special moduli sets. In this case, irreducible trinomials are selected as channel generating polynomials to improve the multiplier's performance and simplify the conversion circuit.
- Design and introduction of a new partial conversion method, which is

used to simplify the conversion circuits for the PRNS in modular reduction process.

- Introduction of Redundant PRNS (RPRNS) over binary fields, by which error detection and error correction capability is added to the GF(2^m) arithmetic circuits.
- Design and implementation of the smallest reported memory free AES (Advanced Encryption Standard) encryption core over FPGA platform by exploring the use of LUT (look-up table) based shift registers. This design only requires 184 slices on a Xilinx Spartan 3 (XC3S50) device, and 80 slices on a Spartan 6 (XC6SLX4) device while achieving throughputs of 36.5Mbps and 58.13Mbps respectively.
- Application of the PRNS to the AES. An 8-bit data path AES and a 32-bit data path AES are constructed to analyse the overhead that is brought by the PRNS architecture.
- Application of the RPRNS architecture to the AES to provide multiple error detection capability. Design of a PRNS based Sbox table look-up method.
- Design and implementation of the world's first PRNS based error-detecting AES encryption core. This design is capable of detecting 100% channel errors and 93.75% multiple errors that may occur cross different PRNS channels with an overhead of 58%. In addition, the RPRNS architecture provides this design an intrinsic resistance against probing attacks and higher level of confusion. To the author's knowledge the 8-bit PRNS AES design is known as the smallest AES scheme with multiple error detection capability over FPGA platforms.

1.3 List of Papers

- J. Chu, M.Benaissa, GF(2^m) Multiplier using Polynomial Residue Number System. *IEEE APCCAS 2008*, 30 Nov-4 Dec, Macao, China.
- J. Chu, M. Benaissa, "Polynomial residue number system GF(2^m) multiplier using trinomials", *17th European Signal Processing Conference*, August 24-28 2009, Glasgow Scotland.
- J. Chu, M. Benaissa, "A Novel Architecture of Implementing Error Detecting AES using PRNS", *The 14th Euromicro Conference on Digital System Design*, August 31st to September 2nd, 2011, Oulu, Finland.
- J. Chu, M. Benaissa, "Error Detecting AES using Polynomial Residue Number Systems", accepted by *special issue of the Microprocessors and Microsystems*: Embedded Hardware Design.
- J. Chu, M. Benaissa, "Low Area Memory-free FPGA Implementation of the AES Algorithm", accepted by *The International Conference on Field Programmable Logic and Applications FPL 2012,* August 2012, Oslo, Norway.
- J. Chu, M. Benaissa, "Low Area Error Detecting AES", submitted for review, *IEEE TCAS-2*.

1.4 Overview of Thesis

The contents of this thesis can be summarized as follows:

- Chapter 2: Provides background theory on GF(2^m) arithmetic and RNS, introduces PRNS and its support theory and a brief introduction of the AES algorithm.
- Chapter 3: Presents the proposed PRNS multiplication algorithm and its implementations. Comparisons of different multiplier architectures are made.
- Chapter 4: Presents the proposed RPRNS error detection and error correction method for GF(2^m) arithmetic. Examples and implementations of using the proposed approach to achieve error detection and error correction in GF(2^m) multiplication are given.
- Chapter 5: Overviews the AES implementations and presents the detailed design information for the proposed low area AES encryption core with the implementation results.
- Chapter 6: Describes the application of RPRNS in the AES to achieve multiple error detection and gives detailed implementation results. Comparisons of different AES architectures are made.
- Chapter 7: Concludes the thesis and presents a direction for further research.

Chapter 2
Background Theory

2.1 Introduction

This chapter gives a brief overview of the background theories and algorithms that will be needed throughout this thesis. It starts with presenting the fundamental theory behind Galois Fields, also known as Finite Fields and the Residue Number Systems before introducing the proposed Polynomial Residue Number Systems over binary fields. To support the proposed AES design, an overview of the AES algorithm is also given.

For reasons of brevity, only information relating to the implementation of the proposed architectures will be presented. Interested readers can follow up on the theories using the references provided. Chapter 2.2 outlines the theory of Galois Fields $GF(2^m)$, with emphasis on its arithmetic and typical arithmetic circuits. Chapter 2.3 introduces the theories behind RNS and PRNS with corresponding architecture's properties and their applications. Chapter 2.4 describes the basic operations of the AES algorithm.

2.2 Galois Fields Theory

Galois Fields Theory has been widely used in modern communication and electronic systems. For example in:

- Error-control coding, e.g. Reed-Solomon codes [5]
- Cryptographic schemes, e.g. the Rijndael Algorithm for the AES [6] and ECC over binary field [7]
- Digital signal processing [8]
- Random number generation [9]
- VLSI testing [10]

The basis of Galois Fields is constructed from an algebraic system, called a Group. It consists of a set G, where an operation $\circ$ defined on G satisfying the following properties [11]:

- Abelian: for $x, y \in G$, $(x \circ y)=(y \circ x)$, then G is said to be an abelian group;
- Associativity: for $x, y, z \in G$, $(x \circ y) \circ z = x \circ (y \circ z)$;
- Identity: in G, there is an element e satisfying $(x \circ e) = (e \circ x) = x$ for all $x \in G$;
- Inverse: in G, there exists an unique element $x^{-1} \in G$, satisfying $x \circ x^{-1} = x^{-1} \circ x = e$;
- Closure: for all the elements in the set G, an operation between any pair of elements will result in another element with in the same group G, for $x, y \in G$, $(x \circ y) \in G$.

A Galois Field is an algebraic system consisting of a finite number of elements. The

finite set F and two defined field operations + (addition) and × (multiplication) have the following properties [11]:

- F is an abelian group with respect to the operation +;
- F with the additive element $\{0\}$ removed is an abelian group with respect to the operation ×;
- Distributivity: for $x, y, z \in F$, $\{x \times (y + z) = (x \times y) + (x \times z)\}$, $\{(x + y) \times z = (x \times z) + (y \times z)\}$ and vice versa.

The order q of the field indicates the number of elements in the field. According to the finite field theory, it states that there exists a finite field of order q, if and only if q is either a prime number or a prime power, and this field is denoted as GF(q). If $q = p^m$, where p is a prime number and m is a positive integer, p is then called the field characteristic and m is named as the extension degree of such field. The Galois Fields with characteristic of '2' is known as binary field and its extension field is denoted as GF(2^m). Such GF(2^m) is of particular interest in this thesis as all the proposed architectures and designs are based on this finite field.

2.2.1 The Galois Field GF(2^m)

The Galois Field GF(2^m) is a finite field with the characteristic of '2' and the extension degree of m. It can be viewed as a vector space of dimension m over GF(2) [12]. An element of GF(2^m) A can be denoted uniquely in a vector format as $\{\alpha_{m-1}, \alpha_{m-2}, \cdots, \alpha_1, \alpha_0\}$:

$$A = \sum_{i=0}^{m-1} a_i \alpha_i \ , where\ a_i \in \{0, 1\}$$

The vector $\{\alpha_{m-1}, \alpha_{m-2}, \cdots, \alpha_1, \alpha_0\}$ is called a basis of GF(2^m) over binary field.

2.2.2 GF(2^m) Representations

There are several methods to represent an element over Galois Field. The two most commonly used representations of GF(2^m) elements are using the polynomial basis and the normal basis [13, 14].

- Polynomial Basis (PB or standard basis)

  The field GF(2^m) is generated using an irreducible polynomial of degree m over GF(2), written as:

$$F(x) = x^m + \sum_{i=0}^{m-1} f_i x^i, where\ f_i \in GF(2)$$

  For each irreducible polynomial, there exists a polynomial basis representation, where an element of the defined GF(2^m) field can be uniquely mapped to a binary polynomial of degree less than m. If an element $A \in$ GF(2^m), it can be represented by a polynomial as:

$$A = \sum_{i=0}^{m-1} a_i x^i = a_{m-1}x^{m-1} + \cdots + a_1 x + a_0, where\ a_i \in \{0, 1\}$$

  Usually, A can be denoted by a m-bit bit vector using the coefficients of the above polynomial as $\{a_{m-1}, a_{m-2}, \cdots, a_1, a_0\}$. There exists a smallest positive integer n such that $A^n = 1$, then the n is defined as the order of an element A in the field GF(2^m). If $n = 2^m - 1$, then A is known as a primitive element, where the polynomial basis is given by the set $\{1, A, A^2, \cdots, A^{m-1}\}$, and all other non-zero elements of such field can be generated by $(2^m - 1)$ consecutive powers of the primitive element A.

  Polynomial basis representation is by far the most versatile representation

since it is able to offer appropriate solutions to most computational problems [13]. A detailed example of polynomial basis representation is given in Table 2-1.

Table 2-1: Representations of the elements of GF(2^4) that is generated by the irreducible polynomial $F(x) = x^4 + x + 1$

Power Representation	Polynomial Representation	Bit Vector Representation
0	0	0000
$A^0 = 1$	1	0001
A^1	x	0010
A^2	x^2	0100
A^3	x^3	1000
A^4	$x + 1$	0011
A^5	$x^2 + x$	0110
A^6	$x^3 + x^2$	1100
A^7	$x^3 + x + 1$	1011
A^8	$x^2 + 1$	0101
A^9	$x^3 + x$	1010
A^{10}	$x^2 + x + 1$	0111
A^{11}	$x^3 + x^2 + x$	1110
A^{12}	$x^3 + x^2 + x + 1$	1111
A^{13}	$x^3 + x^2 + 1$	1101
A^{14}	$x^3 + 1$	1001
$A^{15} = A^0 = 1$	1	0001

- Normal Basis (NB)

  If β is a primitive element of GF(2^m), in another word $\beta^{2^m-1} = 1$, the normal basis of this field is of the form $\{\beta, \beta^2, \cdots, \beta^{2^{m-1}}\}$. Each element A in the field GF(2^m) can be written as:

$$A = \sum_{i=0}^{m-1} a_i \beta^{2^i}, where\ a_i \in \{0, 1\}$$

  Squaring operation of an element can be easily implemented by a simple cyclic shift of the coordinates of the normal basis representation; however, the multiplication in the normal basis is more complicated than using polynomial

basis. For reasons of brevity, the detailed information on normal basis representation refers to [12].

As in this thesis, most of the algorithms and implementations are using polynomial basis representation. In the following section, polynomial basis is used to demonstrate the basic arithmetic over GF(2^m). For the arithmetic using other basis, interested reader can refer to [13, 14].

2.2.3 GF(2^m) Arithmetic

The basic arithmetic over GF(2^m) that is being used in this thesis includes: addition, multiplication, squaring, modular reduction and inversion.

The addition and subtraction in GF(2^m) can be implemented as bitwise XOR operation, which performs the modular 2 operation, because the field characteristic is '2'. A very attractive property can be revealed here: the addition operation will not generate a carry signal, which means the lower bits will not affect the higher bits while doing additions. The property will be further exploited and applied to the proposed partial conversion method.

The crux of GF(2^m) arithmetic is the multiplication. It can be demonstrated as the following equation: Let $A, B \epsilon$ GF(2^m), then their product C

$C = AB \ mod \ f(x)$, where $f(x)$ is the generating irreducible polynomial of the field.

f(x) is used to perform degree reduction to ensure that C is also in GF(2^m) and the multiplication is closed. The multiplication algorithm is expressed as in Algorithms 2-1, which is also known as "shift-and-add" algorithms [52, 53, 54]:

Input: $A(x), B(x) \in \mathrm{GF}(2^m)$, irreducible polynomial $F(x)$ of degree m

Output: $C(x) = A(x) \cdot B(x) \; mod \; F(x)$

1: $C(x) \Leftarrow 0$

2: **for** $i = m - 1$ to 0 **do**

$C(x) \Leftarrow C(x) \cdot x + A(x) b_i$

$C(x) \Leftarrow C(x) + F(x) c_m$

end for

3: return $C(x)$

Algorithm 2-1: Multiplication in GF(2^m) with interleaved modular reduction

The typical hardware architecture of implementing GF(2^m) multiplier is usually in one of three ways:

- *Bit-serial multiplication* performs one GF(2^m) multiplication including the modular reduction operation within m clock cycles. The operand is fed bitwisely. The complexity is defined using a linear function O(m), which is relatively low. Typical example can be found in [13, 15]. Figure 2-1 demonstrate a MSB first 4-bit bit-serial multiplier generated using $F(x) = x^4 + x + 1$ over GF(2^4).

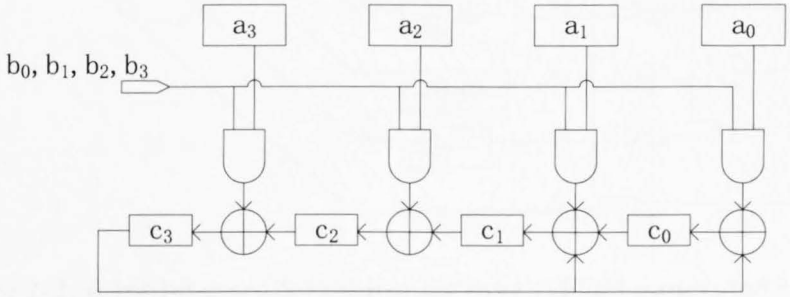

Figure 2-1: 4-bit bit-serial multiplier over GF(2^4) generated by $F(x) = x^4 + x + 1$

- *Bit-parallel multiplication* performs the GF(2^m) multiplication using one clock cycle, or in another word, it can be constructed using pure combinational logic circuit. All the operands are fed in parallel, to reduce the latency of performing such multiplication. The modular reduction process is usually done by multiplying a modular reduction matrix, which is derived from the field generating polynomial. The complexity of this architecture is relatively high compared with bit-serial architecture, hence it is usually defined using a quadratic function O(m^2). Systolic architecture can also be adopted for bit-parallel multiplication in some designs [16]. Further references refer to [17, 18, 19, 20]. A typical bit-parallel multiplier example is shown in Figure 2-2. The AND network and the first level of XOR network calculate the intermediate product *s*; the last level of XORs performs the modular reduction using $F(x) = x^4 + x + 1$ over GF(2^4).

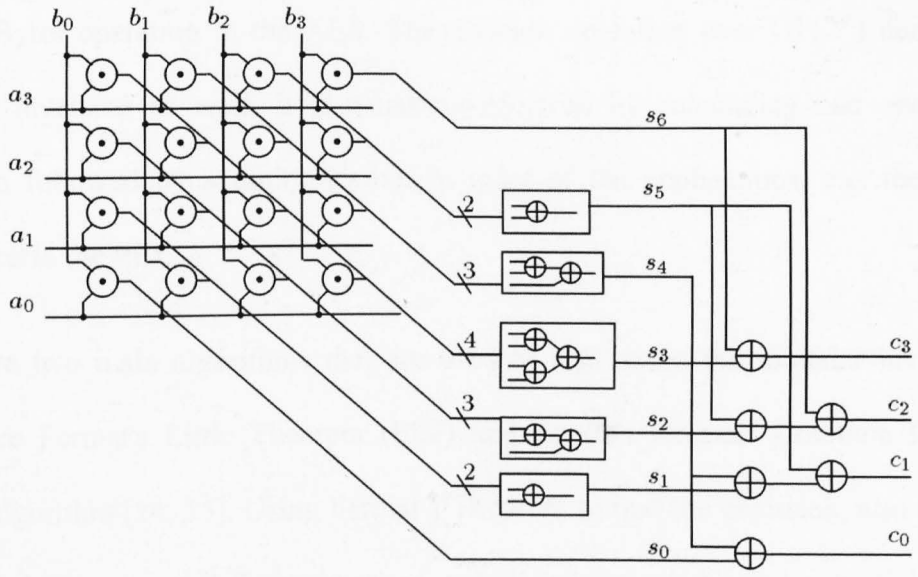

Figure 2-2: 4-bit bit-parallel multiplier over GF(2^4) generated by $F(x) = x^4 + x + 1$

- *Digit-serial/parallel multiplication* is a compromising method between the bit-serial and bit-parallel architecture. By using the adjustable digit/word length, it

reduces the number of clock cycles of the bit-serial multiplier and lowers the area of bit-parallel multiplier [21, 22].

The squaring operation over GF(2^m) is derived by, firstly rerouting the input to generate the intermediate product and then performing the modular reduction. Several efficient architectures are reviewed during the study. In [20], it is presented a pure combinational logic squarer with short critical paths and low complexity. However, it can be only applied for certain Galois fields, which are generated by irreducible trinomials. In [23], the squarer is developed from the multiplication circuits by adding an adapter to reroute the operands. It achieves better speed performance over multiplication alone.

Inversion is the most computationally complex and most expensive operation in hardware to implement among all Galois field arithmetic, however it is a very important operation in the cryptography area mainly because of its nonlinearity, e.g. the SubBytes operation in the AES. The division operation over GF(2^m) needs the modular inversion as well, it is usually performed by calculating one operand's inversion followed by a multiplication in most of the applications, e.g. the point operations in the ECC.

There are two main algorithms that are used to implement the modular inversion, which are Fermat's Little Theorem (FLT) and Euclid's Greatest Common Divisor (GCD) algorithm [24, 25]. Using Fermat's Theorem to find the inversion, also known as the Multiplication and Square method, can be implemented using a multiplication and square chain, which varies according to the used algorithm. Typical square and multiplication algorithm is expressed as following:

For every $\alpha \in GF(2^m)$ and $\alpha \neq 0$,

According to FLT, $\alpha^{2m} = \alpha$

Therefore, $\alpha^{-1} = \alpha^{2m-2}$

Further, $2^m - 2 = 2 + 2^2 + 2^3 + \cdots + 2^{m-1}$

Hence, $\alpha^{-1} = \alpha^2 \cdot \alpha^{2^2} \cdot \alpha^{2^3} \cdots \alpha^{2^{m-1}}$

Algorithm 2-2: Finding Multiplicative Inversion using Fermat's Little Theorem

Input: $A(x) \in GF(2^m)$ and $A(x) \neq 0$, irreducible polynomial $F(x)$ of degree m

Output: $U = A^{-1}(x)$, such that $A^{-1}(x) \cdot A(x) \bmod F(x) = 1$

1: $S \Leftarrow F(x), R \Leftarrow A(x), V \Leftarrow 0, U \Leftarrow 1, delta \Leftarrow 0$

2: **for** $i = 1$ to $2m$ **do**

if $r_m = 0$ **then**

$R \Leftarrow x \cdot R; U \Leftarrow (x \cdot U) \bmod F; delta \Leftarrow delta + 1;$

else

if $s_m = 1$ **then**

$S \Leftarrow S - R; V \Leftarrow (V - U) \bmod F;$

end if

$S \Leftarrow x \cdot S;$

if $delta = 0$ **then**

$R \Leftrightarrow S; U \Leftrightarrow V;$ (exchange polynomials)

$U \Leftarrow (x \cdot U) \bmod F;$

$delta \Leftarrow delta + 1;$

else

$U \Leftarrow (U/x) \bmod F;$

$delta \Leftarrow delta - 1;$

end if

end if

end for

3: **return** U

Algorithm 2-3: Finding Multiplicative Inversion using Euclid's Greatest Common Divisor algorithm [26]

The Euclid's GCD algorithm is initially used to find the greatest common divisor between two numbers; it can be adopted to calculate the modular inversion as well. There are several variations of modified Euclid's algorithm to perform the modular inversion operation. The Algorithm 2-3 demonstrates one of those algorithms [26].

Most GF(2^m) arithmetic can be implemented in the Montgomery domain as well, e.g. the multiplication, the exponentiation and the inversion. Since the Montgomery domain arithmetic is not being used in this thesis, here only lists a few references for the interested reader. Typical papers are [27, 28, 29].

2.3 Residue Number System Theory

有物不知其数，
三三数之剩二，
五五数之剩三，
七七数之剩二。
问物几何？

——《孙子算经》

In the 4th century, the above math puzzle appeared in the Chinese scholar Sun Zi's mathematical manual titled Sun Zi Suanjing (or commonly known as Sun Tzu Suan-ching in RNS literatures [30]). Its English translation is as following [31]:

There are an unknown number of things,
If we count by threes, there is a remainder of 2,
If we count by fives, there is a remainder of 3,
If we count by sevens, there is a remainder of 2,
Find the number of things.

This math puzzle actually described a three-modulus Residue Number System (RNS). The answer to this puzzle is 23. The process of obtaining the answer, which was outlined in this ancient literature, became known as the Chinese Remainder Theorem (CRT) in honour of its Chinese origins. The Greek mathematician Nichomachus is also credited with independently discovering the CRT. The complete solution to the CRT was further developed by another Chinese mathematician named Jiushao Qin in the 13th century [31]. A proof of this theorem was published by Hsin Tai-Wei of the Ming Dynasty of China [32]. Finally, another proof was published by Euler in 1734.

A small group of Czechoslovakian researchers published the first works on residue arithmetic in 1950s, where their study was to explore RNS's natural fault tolerant property to design an RNS computer [33]. After the birth of the transistor-based computer in the 1960s, the research into RNS computer was pushed into the background. The focus of RNS studies then shifted to digital signal processing (DSP)

due to its easy implementation of additions and multiplications. The work carried out until 1986 was collected in an IEEE press compilation of papers [32], which serves as an excellent reference on the historical development of RNS.

Nowadays, RNS has been widely studied and applied in many areas, from digital signal processing to communications. In RNS, a large integer is decomposed into a set of co-independent smaller integers, so that a large calculation can be performed as a number of smaller calculations in parallel. It reduces complexity of the arithmetic units especially when large bit lengths operands are encountered [34]. In addition, the digit-parallel property of RNS has the advantage of consuming less switching power, which is the main component of power dissipation for current technological processes [35, 42, 43, 44]. This is an important characteristic for portable and wireless devices.

In the field of cryptography, the parallel independent nature of RNS provides a different dimension to data randomization [35], which has been advocated for combating side-channel analysis, such as: simple/differential power analysis, simple/differential electromagnetic analysis and probing attack [3, 36]. The nature of fault tolerance provided by the RNS, which has already been widely applied in DSP and communication systems, is a great attraction for implementing error free crypto-systems, which is useful to fight against fault induction attack.

2.3.1 Residue Number System Representation [35]

RNS is a non-weighted number system defined by a base, which is constructed by N-tuple of positive integers: $\langle m_0, m_1, \cdots, m_{N-1} \rangle$, known as the moduli of the system [37]. As it can be seen from the above notation, the base of RNS consists multiple radixes, unlike a fixed-radix number system (e.g. decimal, binary, etc.).

For a given base, the maximum representational coverage (also known as the dynamic range) of such RNS is defined by the least common multiple (LCM) of the moduli, and is denoted by:

$$M = \text{LCM}\{m_0, m_1, \cdots, m_{N-1}\}$$

Hence, for maximum representational efficiency in RNS, it is imperative for all moduli to be relatively prime, which is denoted as:

$$^1\text{GCD}(m_i, m_j) = 1, \text{for } i \neq j, \text{where } i, j \in [0, N-1]$$

In this case, the dynamic range of the given RNS is:

$$M = \prod_{i=0}^{N-1} m_i$$

If an unsigned integer X stays in the dynamic range $[0,\ M-1]$, it can be represented uniquely in the defined RNS by using its remainders:

$$X \xrightarrow{RNS} \{x_0, x_1, \cdots, x_{N-1}\} \text{ where } x_i = X \bmod m_i, \text{ for } i = 0,1,2, \cdots, N-1$$

If the RNS is used to present a signed integer, the dynamic range is then divided into positive and negative regions. It is defined as:

$$\begin{cases} \left[-\dfrac{M-1}{2}, \dfrac{M-1}{2}\right] & \text{for } M \text{ is odd} \\ \left[-\dfrac{M}{2}, \dfrac{M}{2}-1\right] & \text{for } M \text{ is even} \end{cases}$$

Negative values are mapped into the upper-half of the interval $[0, M-1]$ in RNS. A

[1] GCD represents the greatest common divisor

negative integer is congruent to its additive inverse, which is described algebraically as:

$$|-X|_M \equiv |M - X|_M$$

An example of the $\langle 3, 5 \rangle$ RNS to represent the unsigned and signed numbers is demonstrated in the following table:

Table 2-2: RNS representations of unsigned and signed integers

Signed	Unsigned	$\langle 3, 5 \rangle$ RNS	Signed	Unsigned	$\langle 3, 5 \rangle$ RNS
0	**0**	$\{0, 0\}$	**-7**	**8**	$\{2, 3\}$
1	**1**	$\{1, 1\}$	**-6**	**9**	$\{0, 4\}$
2	**2**	$\{2, 2\}$	**-5**	**10**	$\{1, 0\}$
3	**3**	$\{0, 3\}$	**-4**	**11**	$\{2, 1\}$
4	**4**	$\{1, 4\}$	**-3**	**12**	$\{0, 2\}$
5	**5**	$\{2, 0\}$	**-2**	**13**	$\{1, 3\}$
6	**6**	$\{0, 1\}$	**-1**	**14**	$\{2, 4\}$
7	**7**	$\{1, 2\}$	**-8**	**15**	out of range

2.3.2 Residue Number System Arithmetic

The most basic arithmetic in RNS includes: addition, subtraction, multiplication and the conversion to and from RNS. Let $\circ$ denote the operation of addition, subtraction or multiplication. If X, Y and their result Z (as the operations in RNS are closed) are in the dynamic range $[0, M - 1]$, that is defined by the given RNS, the operation between X and Y to obtain Z is expressed as following: For X, Y and Z in their RNS format respectively as $\{x_0, x_1, \cdots, x_{N-1}\}$, $\{y_0, y_1, \cdots, y_{N-1}\}$ and $\{z_0, z_1, \cdots, z_{N-1}\}$, to obtain $Z = X \circ Y$ in RNS:

$$Z \xrightarrow{RNS} \{z_0, z_1, \cdots, z_{N-1}\} \text{ where } z_i = (x_i \circ y_i) \bmod m_i, i \in [0, N - 1]$$

Note that z_i is solely dependent upon x_i and y_i from the above equation; hence the

RNS operation can be performed in parallel without any data dependency between different RNS channels. The operands used for the RNS operation are the remainders of the original data, those residue digits have smaller bit length compared with the normal weighted operands. This property is often referred to as the "carry-free" property, but this is somewhat misleading since carries may still exist in computations involving residue digits, as in each residue channel it uses weighted number system. The fact is that, in the RNS, carries have not totally disappeared, however the carry propagation delay has been cut short due to using smaller bit length operands, which avoids the main temporal constraint in traditional arithmetic implementation [38]. As a direct consequence of this property, RNS architecture is capable of performing faster addition and multiplication relative to equivalent two's complement operations [35].

Observe from the above equation, a mod operation is needed in each channel's $\circ$ operation. To perform the mod operation, modular arithmetic circuits are needed to be constructed for different modular operations, e.g. modular adder, modular multiplier, etc. The implementation of the modular arithmetic circuits is a big area of research and has been widely studied. Since in this thesis most of the modular operations are not in the integer domain, due to the reason of brevity, only some work that is related to the implementation of the modular arithmetic is listed for interested readers. Typical works can be found in [30, 32, 35, 38].

2.3.3 Residue Number System Converter

To apply the RNS to normal weighted number systems, it is necessary to build the converter to and from the RNS. Moreover, due to the fact that the magnitude and sign determination cannot be performed directly from RNS, and in order to prevent

overflow or to perform error detection in some operations, a conversion back to normal representation seems crucial.

The conversion to RNS is simple in math, which is just a mod operation. However, the conversion from RNS back to normal weighted number system is complicated and expensive in hardware. This conversion is made possible by the CRT, where it states: for a given group of co-prime positive integers $m_0, m_1, \cdots, m_{N-1}$, there exists an integer X satisfying the following system of simultaneous congruence:

$$\begin{aligned} X &\equiv x_1 \pmod{m_1} \\ X &\equiv x_2 \pmod{m_2} \\ &\vdots \\ X &\equiv x_{N-1} \pmod{m_{N-1}} \end{aligned}$$

Furthermore, all solutions X of this system are congruent modulo to the product $M = \prod_{i=0}^{N-1} m_i$.

There are two commonly used algorithms to obtain the solution X, in other words, to convert or to decode the residue representations. They are the CRT and the Mixed Radix Conversion (MRC).

Below are various equivalent algebraic representations of the CRT [35, 37, 39]

$$X = \left| \sum_{i=0}^{N-1} \widehat{m}_i \left| \frac{x_i}{\overline{m}_i} \right|_{m_i} \right|_M$$

$$X = \left| \sum_{i=0}^{N-1} \widehat{m}_i \left| \frac{1}{\overline{m}_i} \right|_{m_i} x_i \right|_M$$

$$X = \sum_{i=0}^{N-1} \widehat{m}_i \left| \frac{x_i}{\overline{m}_i} \right|_{m_i} - qM$$

where $M = \prod_{i=0}^{N-1} m_i$, $\widehat{m}_i = \frac{M}{m_i}$, $\left| \frac{x_i}{\overline{m}_i} \right|_{m_i}$ is the multiplicative inverse of $\widehat{m}_i \bmod m_i$, x_i

is the residue representation of X, and where $q \in [0,\ N-1]$.

It is observed that, the operations in CRT are performed in parallel, which is attractive property for fast designs. However, the major difficulty of implementing it is the final mod M operation, as M can be a large and arbitrary number. Several innovative solutions have appeared in the literature that aims to address this problem. Readers can refer to [35, 40, 41] for more information.

To use MRC to calculate the value of X, firstly, the residue representation has to be translated to a mixed-radix representation. If the mixed-radix number system (MRS) is combined with the RNS, obtaining X can be expressed as:

$$X = a_{N-1}\prod_{i=0}^{N-2} m_i + \cdots + a_2 m_1 m_0 + a_1 m_0 + a_0$$

where the m_i are the moduli of the RNS and the a_i's $(0 \le a_i \le m_i)$ are the mixed radix digits (MRDs). The classic algorithm to obtain the MRDs $(a_i's)$ is a sequential process that was proposed by Szabo and Tanaka [37], expressed as:

$$Y_i = \frac{Y_{i-1} - a_{i-1}}{m_{i-1}}$$

$$a_i = |Y_i|_{m_i}$$

where $Y_0 = X, a_0 = x_0$ and $i \in [1,\ N-1]$.

In contrast to the CRT, the derivation of the MRSs requires *N-1* stages, which consumes more time while the number of moduli increases. However, it is useful to note that the arithmetic operations in the MRC algorithm are processed using residue hardware and the mod M operation is avoided.

The proposed PRNS over binary field shares most of the characteristics, arithmetic and algorithms with the RNS, it is going to be discussed in the next section.

2.4 Polynomial Residue Number System over GF(2^m)

PRNS were first used to achieve better performance in signal processing with a high degree of parallelism [45, 46, 47]. A PRNS over GF(2^m) that is similar to normal RNS over integers, was firstly introduced in [48] to construct a GF(2^m) multiplier. In the PRNS, each channel is generated by a polynomial instead of a prime number as in the typical RNS. The Chinese Remainder Theorem (CRT), which is valid in RNS, can also be applied to PRNS [1].

2.4.1 Polynomial Residue Number System Representation

A list of irreducible polynomials over binary fields is selected as the field generating polynomials for PRNS channels. The list is written in polynomial representation as: $m_1(x), m_2(x), \cdots, m_N(x)$, where N is the number of channels. The degree of each m_i is d_i. The dynamic range of the given PRNS is constrained by the product polynomial $M(x)$, denoted as:

$$M(x) = \prod_{i=1}^{N} m_i(x)$$

In order to represent an arbitrary GF(2^m) element uniquely using its residues, the degree D of the product polynomial $M(x)$ should be no less than m, that is

$$D = \sum_{i=1}^{N} d_i \geq m$$

If the PRNS is used for GF(2^m) multiplication, the dynamic range covered by the selected PRNS should satisfy the following inequation, because it should cover the intermediate product of two arbitrary field elements:

$$D = \sum_{i=1}^{N} d_i \geq 2m$$

Within the dynamic range, then a polynomial basis field element $p(x)$ can be represented uniquely in PRNS format using a list of its polynomial remainders:

$$\vec{p}(x) = \{p_1(x), p_2(x), \ldots, p_N(x)\}$$

Where $\vec{p}(x) = p(x) \; mod \; m_i(x)$ for $i = 1, 2, \ldots, N$.

An example is given in Table 2-3 to demonstrate the PRNS representation. In this example, $x^2 + x + 1$ and $x^3 + x + 1$ are selected as the residue channel generating polynomial $m_1(x)$ and $m_2(x)$ respectively. The given moduli set is being used to represent an arbitrary element over GF(2^4):

Table 2-3: PRNS representations of GF(2^4) elements

GF(2^4) element in PB	GF(2^4) element in binary vector	PRNS Representation in PB	PRNS Representation in binary vector
0	0000	$\{0,0\}$	{00,000}
1	0001	$\{1,1\}$	{01,001}
x	0010	$\{x,x\}$	{10,010}
$x+1$	0011	$\{x+1,x+1\}$	{11,011}
x^2	0100	$\{x+1,x^2\}$	{11,100}
x^2+1	0101	$\{x,x^2+1\}$	{10,101}
x^2+x	0110	$\{1,x^2+x\}$	{01,110}
x^2+x+1	0111	$\{0,x^2+x+1\}$	{00,111}
x^3	1000	$\{1,x+1\}$	{01,011}
x^3+1	1001	$\{0,x\}$	{00,010}
x^3+x	1010	$\{x+1,1\}$	{11,001}
x^3+x+1	1011	$\{x,0\}$	{10,000}
x^3+x^2	1100	$\{x,x^2+x+1\}$	{10,111}
x^3+x^2+1	1101	$\{x+1,x^2+x\}$	{11,110}
x^3+x^2+x	1110	$\{0,x^2+1\}$	{00,101}
x^3+x^2+x+1	1111	$\{1,x^2\}$	{01,100}

2.4.2 Polynomial Residue Number System Arithmetic

In PRNS over GF(2^m), as in normal RNS, the most commonly used arithmetic is addition/subtraction and multiplication. Those operations can be performed in parallel as: for A, B and their results are all covered by the given dynamic range

$$A \pm B = \{\langle a_1 \; XOR \; b_1 \rangle_{m_1}, \ldots, \langle a_N \; XOR \; b_N \rangle_{m_N}\}$$

$$A \times B = \{\langle a_1 \times b_1 \rangle_{m_1}, \ldots, \langle a_N \times b_N \rangle_{m_N}\}$$

where a_i and b_i for $i \in [1, N]$ are the PRNS representation of A and B.

Due to the fact that addition and subtraction operations are performed by bitwise XOR in binary field (which performs mod 2 operation), there exists no overflow problems, hence modular reduction using $m_i(x)$ is not needed in addition and subtraction operations. However the channel multiplication's mod $m_i(x)$ operation is not avoidable, since the arithmetic of each residue channel is over GF(2) as well, basic GF(2) multiplication circuit can be used to implemented the residue channel multiplier.

It has to be noticed that the modular reduction over GF(2^m) is still necessary for multiplications to ensure all operations are closed. Since the magnitude determination cannot be performed directly from PRNS, and in order to prevent overflow, a conversion back to polynomial representation is necessary before performing the GF(2^m) modular reduction. Meanwhile, the conversion is also required by the error detection to check if there is any overflow.

The conversion to PRNS can be implemented straight forward using GF(2) modular reduction circuits. The conversion from PRNS format to weighted polynomial

representation is based on the extension of the CRT to polynomials. The Single Radix Conversion (SRC) algorithm is introduced to perform the conversion in this thesis. It is described as [48]:

$$p(x) = \sum_{i=1}^{N} [p_i(x) \cdot I_i(x) \bmod m_i(x)] \cdot M_i(x)$$

$$\text{Where } M_i(x) = \frac{M(x)}{m_i(x)} = m_1(x) \ldots \cdot m_{i-1}(x) \cdot m_{i+1}(x) \ldots \cdot m_N(x)$$

$$I_i(\mathrm{x}) = M_i^{-1}(\mathrm{x}) (\bmod m_i(\mathrm{x}))$$

$I_i(x)$ is the multiplicative inversion of $M_i(x)$ mod m_i. Usually, in the implementation, $I_i(x)$ and $M_i(x)$ are pre-calculated according to the given PRNS.

The SRC algorithm is the extension of the CRT algorithm to binary field. Due to the carry-free property in binary field, the final (mod M) operation, which exists in CRT for integers [35, 37, 39], is not necessary in SRC over binary field. A detailed example is given in Appendix B to demonstrate the SRC algorithm.

2.5 The Advanced Encryption Standard

The Rijndael cipher algorithm, introduced by Vincent Rijmen and Joan Daemen was selected as the Advanced Encryption Standard (AES) by the National Institute of Standards and Technology (NIST) in 2000. In the following year, this algorithm became the Federal Information Processing Standard FIPS-197. The reader is referred to FIPS-197 [6], which is the original official documents of the AES algorithm, for a detailed description.

The AES is a symmetric block cipher, which uses the same key for both encryption and decryption. It has been broadly used for different applications, including smart cards and cellular phones, website servers and automated teller machines etc. Similar to other symmetric cyphers, the AES applies round operations iteratively to the plaintext to generate the ciphertext. Operations in the Rijndael cipher are defined over the $GF(2^8)$ in the polynomial basis with a non-primitive irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$. In line with the original document, this section follows the notation and definition of terms in FIPS-197 [6].

The round operations are applied to a 128-bit state, which is organized into 4 columns of 4 bytes (in total 16 bytes). The cipher key can have three different bit sizes 128, 192 or 256 bits to achieve different level of security; it is also organized into N_k (4, 6 or 8) columns of 4 bytes each. The number of round operations, denoted as N_r, is determined by the length of the cipher key N_k (shown in Table 2-4). The round operation consists of four sub-transformations: SubBytes, ShiftRow, MixColumn and AddRoundKey (shown in Algorithm 2-4). Derived from the cipher key, each round key is generated by an extra KeyExpansion function.

Table 2-4: Number of Round Transformations with respect to N_k

Key Length	N_r
$N_k = 4$ (128 bits)	10
$N_k = 6$ (192 bits)	12
$N_k = 8$ (256 bits)	14

```
RoundTransform (State, RoundKey)
{       SubBytes  (State);
        ShiftRow  (State);
        MixColumn  (State);
        AddRoundKey  (State, RoundKey);  }
```

Algorithm 2-4: AES Round Transformations

The full AES algorithm (encryption process) is described in Algorithm 2-5:

```
AES (Plaintext, CipherKey)
{-- Initializing
        RoundKey = CipherKey;
        State = Plaintext;
 -- Add the original key
        AddRoundKey  (State, RoundKey);
 -- Round Transformation
        for I in 1 to Nr – 1 loop
                RoundKey = KeyExpansion (RoundKey, RC);
                RoundTransform (State, RoundKey);
        End loop;
 -- Final Round
        RoundKey = KeyExpansion (RoundKey, RC);
        SubBytes  (State);
        ShiftRow  (State);
        AddRoundKey  (State, RoundKey);
        Output = State;
        }
```

Algorithm 2-5: AES Algorithm (Encryption)

It has to be noticed that, each round transformation contains four sub operations and a RoundKey computation, with the exception of the final round, where the only difference in the last round is the absence of a MixColumn operation. The decryption process using the AES performs the encryption process in a straightforward reversed order. The sub round operations are explained individually in details in the following

sections. For the reason of convenience, the 128-bit State and RoundKey are denoted using a 4×4 matrix as:

$$\text{State: } \begin{vmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{vmatrix}, \text{ RoundKey: } \begin{vmatrix} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} \\ k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} \\ k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} \\ k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} \end{vmatrix}$$

Figure 2-3: State and RoundKey Notation

2.5.1 SubBytes and InvSubBytes

The SubBytes transformation is the only non-linear operation among all AES transformations. It substitutes every byte of an AES State to another byte. The forward SubBytes operation is used for the encryption and the inverse SubBytes is for the decryption process. The substitution follows the following rule: the forward SubBytes operation applies a Forward Affine Transformation[1] to a multiplicative inverse of a byte; the inverse SubBytes, denoted as InvSubBytes, plays an Inverse Affine Transformation to a byte first, then computes its multiplicative inversion. It can be expressed as:

SubBytes:

$$s'_{i,j} = \text{Forward Affine Transformation (Multiplicative Inverse } (s_{i,j}))$$

InvSubBytes:

$$s'_{i,j} = \text{Multiplicative Inverse (Inverse Affine Transformation } (s_{i,j}))$$

The affine transformation can be expressed as:

$$b'_i = b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus c_i$$

[1]Affine Transformation is a reversible linear transformation as $y = ax + b$. In the AES, equation $y = ax + b$ is expressed in matrix.

for $0 \leq i \leq 8$ and b_i is the i^{th} bit of a byte c_i is the i^{th} bit of the byte with hexadecimal value $[63]_{\text{HEX}}$.

or using matrix form as:

$$\begin{bmatrix} b_0' \\ b_1' \\ b_2' \\ b_3' \\ b_4' \\ b_5' \\ b_6' \\ b_7' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

2.5.2 ShiftRow and InvShiftRow

The ShiftRow transformation cyclically shifts the rows of the state with different number of bytes according to the row number. Row 0 is not shifted, Row 1 is shifted by 1 byte, Row 2 is shifted by 2 bytes and Row 3 is shifted by 3 bytes. The ShiftRow shifts the byte cyclically towards left in the encryption process, while shifts to the opposite direction in the decryption process, denoted as InvShiftRow. The shifting process is demonstrated as Figure 2-4:

$$\begin{vmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,3} & s_{1,0} & s_{1,1} & s_{1,2} \\ s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3,1} & s_{3,2} & s_{3,3} & s_{3,0} \end{vmatrix} \xleftarrow{\text{InvShiftRow}} \begin{vmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{vmatrix} \xrightarrow{\text{ShiftRow}} \begin{vmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,1} & s_{1,2} & s_{1,3} & s_{1,0} \\ s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3,3} & s_{3,0} & s_{3,1} & s_{3,2} \end{vmatrix}$$

Figure 2-4: ShiftRow Operation for Encryption and Decryption

2.5.3 MixColumn and InvMixColumn

The MixColumn transformation, just as its name implies, is a column based operation to the State. It treats each column as a four-term polynomial and performs

multiplication with a fixed polynomial $a(x)$ followed by a modulo $x^4 + 1$ operation. The $a(x)$ is given by:

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$$

For easier implementation, the polynomial multiplication is usually written in matrix format. See the following equation, it shows the MixColumn transformation of the i^{th} column of a State using matrix multiplication:

$$\begin{bmatrix} s_{0,i}' \\ s_{1,i}' \\ s_{2,i}' \\ s_{3,i}' \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} s_{0,i} \\ s_{1,i} \\ s_{2,i} \\ s_{3,i} \end{bmatrix}$$

Due to the fact that the polynomial $a(x)$ is co-prime to $x^4 + 1$, there exists its inversion $a^{-1}(x)$. It is calculated as:

$$a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\}$$

In the decryption process, the Inverse MixColumn transformation (written as InvMixColumn), multiplies a column with the polynomial $a^{-1}(x)$, then performs a modulo $x^4 + 1$ operation. In matrix format, the InvMixColumn is demonstrated as:

$$\begin{bmatrix} s_{0,i}' \\ s_{1,i}' \\ s_{2,i}' \\ s_{3,i}' \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \cdot \begin{bmatrix} s_{0,i} \\ s_{1,i} \\ s_{2,i} \\ s_{3,i} \end{bmatrix}$$

2.5.4 AddRoundKey

The AddRoundKey transformation performs the addition over GF(2^8) of each byte in a State with a suitable RoundKey that is generated by the KeyExpansion function (see

Figure 2-5). The addition over binary field can be implemented using bit-wise XOR operation, as it acts as a modular 2 adder. The AddRoundKey transformation is identical for both encryption and decryption process.

$$\begin{vmatrix} s_{0,0}' & s_{0,1}' & s_{0,2}' & s_{0,3}' \\ s_{1,0}' & s_{1,1}' & s_{1,2}' & s_{1,3}' \\ s_{2,0}' & s_{2,1}' & s_{2,2}' & s_{2,3}' \\ s_{3,0}' & s_{3,1}' & s_{3,2}' & s_{3,3}' \end{vmatrix} = \begin{vmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{vmatrix} \oplus \begin{vmatrix} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} \\ k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} \\ k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} \\ k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} \end{vmatrix}$$

Figure 2-5: AddRoundKey Transformation

2.5.5 KeyExpansion

The KeyExpansion (or known as KeySchedule in some works) produces the RoundKeys that are used in the AddRoundKey operation. The total number of RoundKeys in columns is equal to the block length multiplied by the number of rounds. A new RoundKey is derived from the RoundKey from the previous round. The KeyExpansion algorithm is shown in Algorithm 2-6:

```
KeyExpansion (RoundKey [0 to 3], RC)
--The RoundKey is organized as 4 columns of 4 bytes each
{-- Initializing
        Rcon = (RC, '00', '00', '00') ;
        temp = RoundKey [3] ;
 -- Key Generating
        temp = SubBytes (RotByte(temp)) XOR Rcon ;
        NextRoundKey [0] = RoundKey [0] XOR temp ;
        NextRoundKey [1] = RoundKey [1] XOR NextRoundKey [0] ;
        NextRoundKey [2] = RoundKey [2] XOR NextRoundKey [1] ;
        NextRoundKey [3] = RoundKey [3] XOR NextRoundKey [2] ;
 -- Calculating RC
        RC = (RC · x) mod x^8 + x^4 + x^3 + x + 1 ;
 -- Output
        Return (NextRoundKey, RC) ;
        }
```

Algorithm 2-6: KeyExpansion Algorithm (128-bit Key)

There requires two sub-functions in the above algorithm: RotByte and SubBytes. The

RotByte rearranges the location of four bytes in one column by cyclically shifting the column by 1 byte. The SubBytes function is the same as the SubBytes described in Chapter 2.5.1, although it is on a smaller scale i.e. one column instead of one state. KeyExpansion also contains a calculation of the round constant value R_{con}, it is defined as $R_{con}[i] = (RC[i], '00', '00', '00')$, where i indicates the round number. $RC[i]$ represents an element in $GF(2^8)$ with a value of x^{i-1}. The mod operation in the calculating of R_{con} ensures there is no overflow and all the value stays in $GF(2^8)$.

2.6 Design, Validation and Verification

The proposed designs in this thesis that will be described in the next a few chapters are carried out using VHDL (Very High Speed Integrated Circuit Hardware Description Language) [49, 50]. VHDL together with Verilog, are two of the most popular hardware description languages in today's digital design.

Design validation and verification are realized using a combination of high-level simulation by using ModelSim [51], which is a powerful simulator from Mentor Graphics, and Field Programmable Gate Array prototyping, using the Xilinx ISE design suite. In both cases, synthesizable Register Transfer Level (RTL) VHDL code is written to enable a smooth transition between high-level simulation using ModelSim and FPGA prototyping.

In order to verify the correctness of the proposed architectures and designs, valid testing vectors are generated either from third party open source C libraries or from the official standards (e.g. the testing vectors for the AES from FIPS-197). The testing vectors are run through the design using a ModelSim based test bench in VHDL for functional verification, where the test bench has three main purposes:

- To generate stimuli for simulation
- To apply these stimuli to the design under test (DUT)
- To compare the output of the DUT with the expected values

In addition, to simplify the testing and verification process in some cases, extra testing circuits are built to connect with the DUT to verify the results.

2.7 Research Proposal

After Rijndael cipher algorithm being selected as the AES by the National Institute of Standards and Technology (NIST) in 2000, this crypto-scheme has been widely adopted for various applications from high-end computers to low power portable devices. In recent years, numerous attacks have been introduced to break cryptographic systems and extract secret information via; side-channel-analysis by analysing or manipulating the observations of physical characteristics of the electronic cryptographic system. Typical examples are timing attacks [94], power attacks [95], electromagnetic radiation attacks [96] and fault attacks [97, 98]. Prior work has shown that even a single transient error occurring during the AES round operations will very likely result in a large number of errors in the final data [68]. In addition, most of the attack scenarios have shown that the AES is quite vulnerable to fault attacks [68, 69, 70, 71, 72]. Hence it is necessary to provide error detection mechanisms to the AES design to achieve higher level of reliability and security. There are several approaches to achieve error detection for cryptographic systems. Generic solutions are duplication and repeated computation, however these solutions either double hardware overhead or latency and they are not protective against permanent faults. Error detecting codes are widely used by engineers to implement error proof designs. A good review of the existing error detecting method for the AES can be found in [99], where it summarises two solutions: parity code based schemes [68, 100, 101] and residue code based schemes [102, 103]. The first group of schemes have low overhead but are weak for multiple faults detection; the latter ones yield good multiple error coverage but are weak in single fault detection and become very hardware consuming when predicting the residue codes for non-linear operations such as SubBytes.

Since security is becoming an essential part of modern communication. However, to the author's knowledge, there are rare solutions that can provide full protection against attacks to crypto-schemes, where the solutions are either focused on fault attacks or focused on reducing the information leaking from side-channels. Hence it is necessary to look for an all-in-one solution that can provide the crypto-systems with both fault resistance and side-channel-attack resistance with reasonable hardware overhead and good error coverage. In this research, the RNS architecture over binary field (PRNS) is proposed to achieve such goals for the selected AES scheme. The two main reasons are as follows:

- RNS architecture for error detection and correction has a good balance between error coverage and hardware overhead, which has already been widely applied in DSP and communication systems.

- The parallel independent nature of RNS architecture provides a different dimension to data randomization [35], which has been advocated for combating side-channel analysis, such as: simple/differential power analysis, simple/differential electromagnetic analysis and probing attack [3, 36].

Furthermore, to minimize the hardware overhead and to achieve quick prototyping, the LUT based shift registers that is provided by the latest FPGA technologies is proposed for realising the shifting operations in the AES over FPGA platforms in this research.

Chapter 3
PRNS Multiplication over GF(2^m)

3.1 Introduction

Many researchers have been encouraged by the escalating use of Galois fields GF(2^m) arithmetic in digital signal processing, cryptography, coding theory and computer algebra to investigate different architectures and novel algorithms to advance Galois field circuits. In this chapter, the GF(2^m) multiplication is investigated and a novel GF(2^m) multiplier architecture and corresponding implementation over GF(2^{163}) that is based on the proposed PRNS is presented.

Consider the example application of the proposed multiplier in the context of public key cryptosystems, in particular elliptic curve cryptography (ECC) where the required large operands impose many design challenges. The curve K-163 presented in Fips-186 [7] is chosen as a standardized curve for ECC over the binary field. It uses the field generating polynomial $f(x) = x^{163}+x^7 + x^6 + x^3 + 1$ over GF(2^{163}). A GF(2^{163}) multiplier is constructed to demonstrate the proposed PRNS architecture.

To obtain a good analysis of the PRNS architecture based multiplier, a channel-serial and a channel-parallel PRNS multiplier are constructed individually to compare the synthesis results. To further improve the performance of the propose PRNS multiplier, a set of special moduli, which are all trinomials, is selected as the channel generating polynomials to reduce the complexity of the conversion circuit. Furthermore, a novel approach of performing the modular reduction over GF(2^{163}) using PRNS is proposed

to simplify the modular reduction circuit.

The organization of this chapter is as follows. Firstly, architectural descriptions are provided for the proposed channel-serial and channel-parallel PRNS multiplier over GF(2^m). Thereafter, the derivation of the proposed partial modular reduction method is introduced, followed by the description of the improved PRNS GF(2^m) multiplier. Before conclusions are drawn, the FPGA implementation results are presented and compared between different GF(2^m) multiplier architectures.

3.2 GF(2^{163}) Multiplication using PRNS

3.2.1 Dynamic Range and Moduli Set

To cover the whole dynamic range of two arbitrary field elements' multiplication, 37 9-degree irreducible polynomials are selected as the PRNS channels. This satisfies the inequation $d \times N \geq 2m$, where $d = 9, N = 37, m = 163$. The reasons why 9-degree irreducible polynomials are selected are, firstly, from the exhaustive list of irreducible polynomials [51], the degree 9 irreducible polynomials satisfy the inequation $d \times N \geq 2m$ (Chapter 2.4.1) with the smallest degree. In other words, let's say if degree 8 irreducible polynomials are chosen, from the exhaustive list there exists 30 irreducible polynomials with degree 8, since ($30 \times 8 = 240 < 2 \times 163$), these polynomials cannot cover the whole dynamic range over GF(2^{163}) multiplication and neither do the polynomials with even smaller degree. The second reason is that, smaller degree means shorter operands and shorter channel length; in other words, it will lead to simpler channel circuit.

However, there are trade-offs between the channel length and the number of channels. To cover the same dynamic range, shorter channel length requires more channels, which leads to more parallelism or more cycles according to different channel architecture.

With the given moduli set, assuming the result is $q(x)$, the GF(2^{163}) multiplication is performed as :

$$p(x) = A(x) \times B(x) \xrightarrow{PRNS} \{\langle a_1 \times b_1 \rangle_{m_1}, \ldots, \langle a_{37} \times b_{37} \rangle_{m_{37}}\}$$

$$q(x) = p(x) \bmod f(x), \text{where } f(x) = x^{163} + x^7 + x^6 + x^3 + 1$$

Due to the fact that multiplication over Galois field is closed, to prevent overflow, modular reduction is performed following the calculation of the intermediate product using the field generating polynomial $f(x)$, whose degree is equal to 163. Since the magnitude of the intermediate product cannot be determined directly from the PRNS format, a conversion back to normal polynomial representation is required before performing the modular reduction using $f(x)$. The conversion uses the SRC algorithm that is introduced in Chapter 2.4.2, written according to the chosen moduli set as:

$$p(x) = \sum_{i=1}^{37} (p_i(x) \cdot I_i(x) \bmod m_i(x)) \cdot M_i(x)$$

$$\text{Where } M_i(x) = \frac{M(x)}{m_i(x)} = m_1(x) \ldots \cdot m_{i-1}(x) \cdot m_{i+1}(x) \ldots \cdot m_{37}(x)$$

$$I_i = M_i^{-1}(x) \big(\bmod m_i(x)\big) \text{ for } i \in [1, 37]$$

From the above equation, as the moduli set is predefined, the $M_i(x)$'s and $I_i(x)$'s can be pre-calculated and used as constant value in the conversion implementation. The detailed information on the selection of $m_i(x)$'s and the value of the $M_i(x)$'s and $I_i(x)$'s are listed in Appendix A.

Assuming the input and output data of the proposed multiplier are all in PRNS format, the procedure of performing the GF(2^{163}) multiplication is, firstly performing the channel multiplication, thereafter applying the SRC algorithm to determine the intermediate product $p(x)$, then performing the modular reduction using $f(x)$ to $p(x)$ to obtain $q(x)$, in the end converting the result back to PRNS representation to maintain the consistency.

3.2.2 Channel-Serial PRNS Multiplier over GF(2^{163})

The channel-serial PRNS, just as its name implies, performs the channel operations in serial by sharing a generic channel arithmetic unit. To perform the channel multiplication, a parallel generic GF(2^9) multiplier is constructed, which treats the channel generating polynomials as an input.

To perform the SRC conversion algorithm, $M_i(x)$'s and $I_i(x)$'s are pre-computed and stored into memories. Addresses are used to ensure the pre-stored information is forwarded correctly.

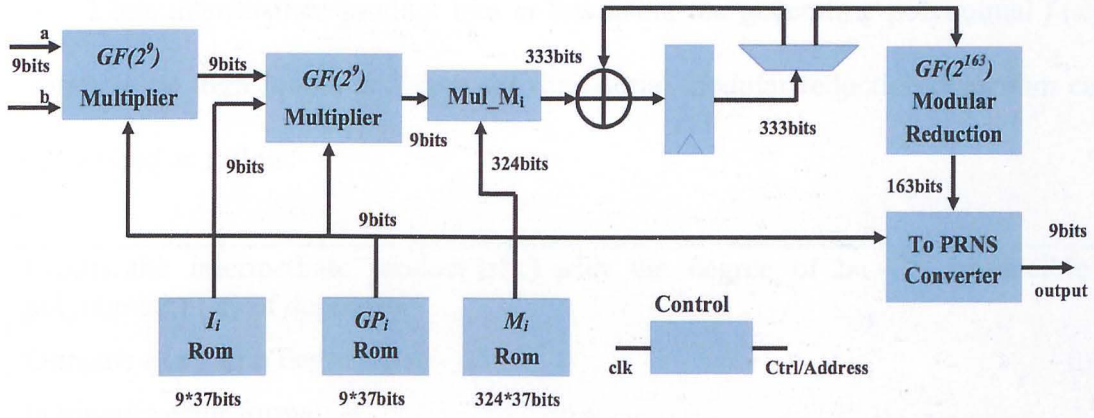

Figure 3-1: Architecture of the Channel-serial PRNS Multiplier over GF(2^{163})

Figure 3-1 shows the full architecture of the channel-serial PRNS multiplier over GF(2^{163}). The GF(2^9) multiplier on the left performs PRNS channel modular multiplication, the one on the right performs part of SRC algorithm which is the multiplication with $I_i(x)$. The block of Mul_M performs the multiplication with $M_i(x)$, it is constructed by pure AND and XOR networks. The following XOR and register calculates the sum of $M_i(x)$, which is the last step of the SRC algorithm. When the SRC algorithm finishes, the register stores the weighted polynomial representation of the intermediate product.

The GF(2^{163}) Modular Reduction block reduces the degree of the intermediate product from 332 to 162. Focusing on polynomial basis multiplications over GF(2^m), several approaches have been reported for the modular reduction, such as "shift-and-add" algorithms [52, 53, 54], look-up-table (LUT) based algorithms [27, 55], Itoh-Tsujii algorithm based reduction method [56], etc.

In this proposed multiplier architectures, a digit-serial modular reduction method based on the conventional iterative reduction scheme is adopted, because it has the capability of providing significant versatility and scalability, which balances the trade-offs between area and operation time [57]. The modular reduction is to reduce the $2m-1$ bits intermediate product into m bits using the generating polynomial $f(x)$. Assuming the digit length is l, then the digit-serial modular reduction algorithm can be described as follow:

Input: the intermediate product $p(x)$ with the degree of $2m-2$, irreducible polynomial $f(x)$ of degree m

Output: $q(x)$ with degree of $m-1$

In binary vector format:

1: $tmp(m-1 \cdots 0) \Leftarrow p(m-1 \cdots 0)$

2: **for** i in 0 to $\left\lfloor \frac{m-1}{l} - 1 \right\rfloor$ **do**

$tmp \Leftarrow tmp$ XOR $\{p[(2m-2-i\cdot l)\cdots(2m-2-i\cdot l-l)]\cdot f(m-1\cdots 0)\}$

end for

3: $q(x) \Leftarrow tmp$

4: **return** $q(x)$

Algorithm 3-1: Digit-serial Modular Reduction Algorithm

Normally, the digit length l is chosen so as $\frac{m-1}{l}$ is an integer to simplify the reduction circuit. Furthermore, the selection of $f(x)$ has the potential of providing significant

simplification in circuit complexity, which is further discussed in the following section.

In this design, the digit length l is selected as 10, such that it needs $\frac{332-162}{10} = 17$ cycles to reduce the degree from 332 to 162. By analysing the generating polynomial of the field, which is $f(x) = x^{163}+x^7 + x^6 + x^3 + 1$, the digit-serial reduction approach is further simplified. By ignoring the MSB in $f(x)$, an 8×10 constant multiplier is used to replace the 163×10 multiplier. This multiplier is constructed by simple AND and XOR networks. Figure 3-2 shows the detailed implementation of the proposed modular reduction:

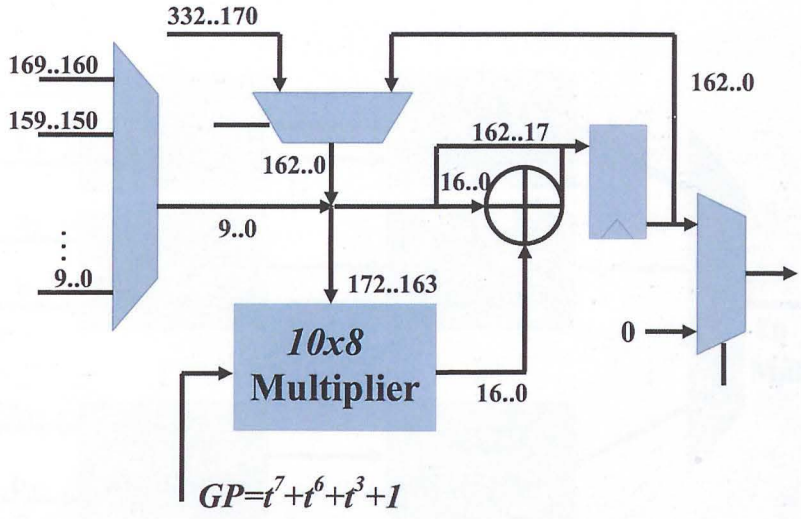

Figure 3-2: Implementation of Digital-serial Modular Reduction Algorithm

The *To_PRNS_Converter* in Figure 3-1 converts the polynomial representation back to PRNS again after the GF(2^{163}) modular reduction. It takes 37 cycles to complete the whole conversion, because the number of channels is 37. The Control unit is simply implemented using a binary counter, which generates the control signals and addresses of the block memory. The entire channel-serial multiplier requires 93 cycles

to perform a multiplication.

In this architecture, the data of each channel is forwarded into and out of the multiplier serially. By sharing the channel modular multipliers, this serial architecture greatly reduces the requirements of hardware resources.

3.2.3 Channel-Parallel PRNS Multiplier over GF(2^{163})

Compared with the channel-serial architecture, the channel-parallel architecture performs the channel modular multiplication in parallel. By modifying the channel-serial architecture, the channel-parallel multiplier is constructed as follows. See Figure 3-3.

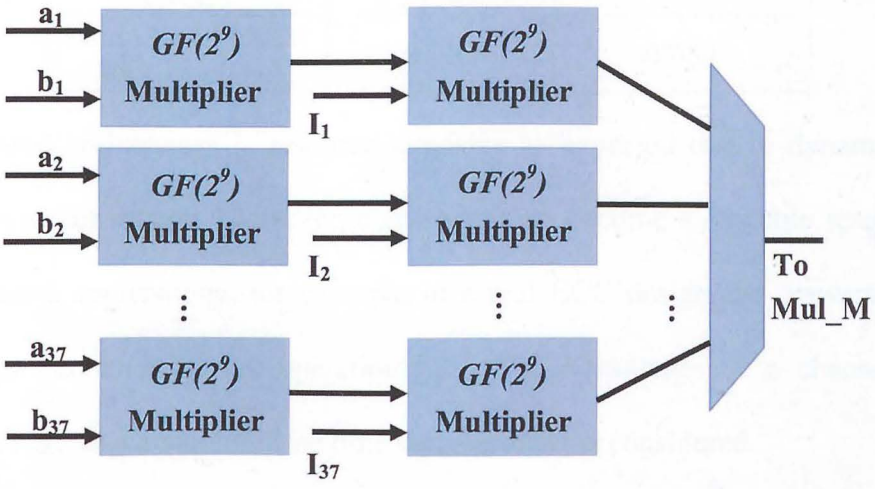

Figure 3-3: Architecture of the Channel-parallel PRNS Multiplier over GF(2^{163})

Following the parallel channel multipliers, the data is forwarded to the *Mul_M* block, after which the circuit is the same with channel-serial architecture. The rest of the architecture which is not shown in Figure 3-3 finishes the SRC algorithm and conduct the to PRNS conversion.

By making comparison with the channel-serial architecture, this architecture uses a

multiple of channel modular multipliers, which increases the hardware cost. However, due to the parallelization of channels, it provides a novel method of parallelization of the multiplication operation over GF(2^m). According to this architecture, 93 cycles is used to perform the multiplication.

3.2.4 Synthesis Results and Comparisons

Xilinx Spartan 3-3s1500lfg320-4 FPGA is used for synthesis and implementation.

Table 3-1: 37-Channel PRNS GF(2^m) Multiplier Synthesis Results

	Channel-serial	Channel-Parallel
FF	1010	1350
LUT	5274	8675
Slices	2752	4625
Frequency (MHz)	5.179	5.119
Cycles	130	93
Time-Area Product (Slices×second)	0.069	0.084

The proposed architecture is resource intensive as expected due to dynamic range coverage and conversion. However, conversion can become a sharable resource for most intended applications, for example in a real ECC design the conversions are required in two ends of the operation [34]. The advantages of a channel serial architecture are also apparent if the time×area product is considered.

From the FPGA synthesis results, it can also be noted that, the speed bottleneck resides in the SRC algorithm and the to-PRNS conversion, resulting in a comparable maximum operating frequency.

To the author's knowledge, there are not any hardware synthesis results for such PRNS architectured GF(2^m) multipliers to compare with.

3.3 Partial Conversion and Modular Reduction using PRNS

The main purpose of introducing such Partial Modular Reduction method is to reduce the complexity of the PRNS conversion and modular reduction circuits, thereby to improve the performance of the PRNS architecture. The feasibility of this method is based on the carry-free property of binary field addition. Let us see an example:

Assume an GF(2^8) element $a(x)$ is represented using polynomial moduli set $\{x^4+x+1, x^4+x^3+1\}$ as $\{a_1(x), a_2(x)\}$ or in binary vector format as $\{\langle a_1(0), a_1(1), a_1(2), a_1(3)\rangle, \langle a_2(0), a_2(1), a_2(2), a_2(3)\rangle\}$. For the given PRNS, to perform the SRC algorithm, the constant values are calculated as:

$$I_1(x) = x^3 + x^2, I_2(x) = x^3 + x + 1$$

$$M_1(x) = x^4 + x^3 + 1, M_2(x) = x^4 + x + 1$$

Let us denote $q_i(x) = [a_i(x) \cdot I_i(x)] \bmod m_i$ and $p_i(x) = q_i(x) \cdot M_i(x)$ for $i = 1, 2$.

Table 3-2: Demonstration of the Partial Conversion Method

	Channel 1	Channel 2
Input	$\langle a_1(0), a_1(1), a_1(2), a_1(3)\rangle$	$\langle a_2(0), a_2(1), a_2(2), a_2(3)\rangle$
$q_i(x)$	$q_1(0) = a_1(1) \oplus a_1(2)$ $q_1(1) = a_1(1) \oplus a_1(3)$ $q_1(2) = a_1(0) \oplus a_1(2)$ $q_1(3) = a_1(0) \oplus a_1(1) \oplus a_1(3)$	$q_2(0) = a_2(0) \oplus a_2(1) \oplus a_2(2)$ $q_2(1) = a_2(0) \oplus a_2(1) \oplus a_2(2) \oplus a_2(3)$ $q_2(2) = a_2(1) \oplus a_2(2) \oplus a_2(3)$ $q_2(3) = a_2(0) \oplus a_2(1) \oplus a_2(3)$
$p_i(x)$	$p_1(0) = q_1(0)$ $p_1(1) = q_1(1)$ $p_1(2) = q_1(2)$ $p_1(3) = q_1(0) \oplus q_1(3)$ $p_1(4) = q_1(1) \oplus q_1(0)$ $p_1(5) = q_1(1) \oplus q_1(2)$ $p_1(6) = q_1(2) \oplus q_1(3)$ $p_1(7) = q_1(3)$	$p_2(0) = q_2(0)$ $p_2(1) = q_2(0) \oplus q_2(1)$ $p_2(2) = q_2(1) \oplus q_2(2)$ $p_2(3) = q_2(2) \oplus q_2(3)$ $p_2(4) = q_2(0) \oplus q_2(3)$ $p_2(5) = q_2(1)$ $p_2(6) = q_2(2)$ $p_2(7) = q_2(3)$
$a(x)$	$a(i) = p_1(i) \oplus p_2(i)$, for $i = 0$ to 7	

$\oplus$ *indicats* XOR *operation*

As $I_i(x)$ and $M_i(x)$ are pre-calculated constant value, multiplying by them can be implemented by using fixed XOR network as demonstrated in Table 3-2.

For example, if we want to determine the Most Significant Bit (MSB) of $a(x)$ from its PRNS representation, according the above table, the computation is straight forward as:

$$a(7) = p_1(7) \oplus p_2(7) = q_1(3) \oplus q_2(3)$$
$$= a_1(0) \oplus a_1(1) \oplus a_1(3) \oplus a_2(0) \oplus a_2(1) \oplus a_2(3)$$

The proposed GF(2^m) modular reduction method for the PRNS is based on the feasibility of the partial conversion. It is derived as follows:

Over GF(2^m), which is generated by $f(x)$, the multiplication is expressed as:

$$pdt(x) = a(x) \cdot b(x) \bmod f(x) \quad (1)$$

The intermediate product, $a(x) \cdot b(x)$ can be expressed in polynomial forms as

$$a(x) \cdot b(x) = c_{2m-2}x^{2m-2} + c_{2m-3}x^{2m-3} + \cdots + c_m x^m + c_{m-1}x^{m-1} + \cdots + c_1 x + c_0$$

Then, equation (1) can be written as

$$pdt(x) = (c_{2m-2}x^{2m-2} + c_{2m-3}x^{2m-3} + \cdots + c_m x^m + c_{m-1}x^{m-1} + \cdots + c_1 x + c_0) \bmod f(x) = (c_{2m-2}x^{2m-2} + c_{2m-3}x^{2m-3} + \cdots + c_m x^m) \bmod f(x) + (c_{m-1}x^{m-1} + \cdots + c_1 x + c_0) \bmod f(x) \quad (2)$$

Since $f(x)$ is a polynomial with degree *m*, then derived from equation (2)

$$pdt(x) = (c_{2m-2}x^{2m-2} + c_{2m-3}x^{2m-3} + \cdots + c_m x^m) \bmod f(x) + (c_{m-1}x^{m-1} + \cdots + c_1 x + c_0) \quad (3)$$

In GF(2), $1 + 1 = 0$, so $(c_{2m-2}x^{2m-2} + c_{2m-3}x^{2m-3} + \cdots + c_m x^m)$ can be added to the left side of the equation (3) twice without changing its value, as following:

$$pdt(x) = (c_{2m-2}x^{2m-2} + c_{2m-3}x^{2m-3} + \cdots + c_m x^m) \bmod f(x) + (c_{m-1}x^{m-1} + \cdots + c_1 x + c_0) + (c_{2m-2}x^{2m-2} + c_{2m-3}x^{2m-3} + \cdots + c_m x^m) + (c_{2m-2}x^{2m-2} + c_{2m-3}x^{2m-3} + \cdots + c_m x^m) \quad (4)$$

Rearranging (4):

$$pdt(x) = (c_{2m-2}x^{2m-2} + c_{2m-3}x^{2m-3} + \cdots + c_m x^m) \bmod f(x) + (c_{2m-2}x^{2m-2} + c_{2m-3}x^{2m-3} + \cdots + c_m x^m) + (c_{2m-2}x^{2m-2} + c_{2m-3}x^{2m-3} + \cdots + c_m x^m + c_{m-1}x^{m-1} + \cdots + c_1 x + c_0)$$

Rewrite the above equation in binary vector format:

$$pdt(x) = (c_{2m-2}, c_{2m-3}, \cdots, c_m, 0, \cdots, 0) \bmod f(x) + (c_{2m-2}, c_{2m-3}, \cdots, c_m, 0, \cdots, 0) + (c_{2m-2}, c_{2m-3}, \cdots, c_m, c_{m-1}, \cdots, c_1, c_0) \quad (5)$$

Assume the result after the mod $f(x)$ operation is donated as $(c'_{m-1}, \cdots, c'_1, c'_0)$, then from the equation (5):

$$pdt(x) =$$

$$(c'_{m-1}, \cdots, c'_1, c'_0) + (c_{2m-2}, c_{2m-3}, \cdots, c_m, 0, \cdots, 0) +$$

$$(c_{2m-2}, c_{2m-3}, \cdots, c_m, c_{m-1}, \cdots, c_1, c_0) =$$

$$(c_{2m-2}, c_{2m-3}, \cdots, c_m, c'_{m-1}, \cdots, c'_1, c'_0) + (c_{2m-2}, c_{2m-3}, \cdots, c_m, c_{m-1}, \cdots, c_1, c_0) \quad (6)$$

Both components in (6) can be expressed using PRNS representation. To obtain that, firstly, it is needed to partially convert the most significant $m-1$ bit of the intermedia product to normal polynomial representation, then perform the modular reduction using $f(x)$ to calculate c'_i $(i \in [0, m-1])$, after which a to PRNS conversion is applied to the combination of c_j and c'_i $(i \in [0, m-1],$ $j \in [m, 2m-2])$ to find its PRNS representation. The second addend of the addition in (6) is in PRNS already, which are the results from the PRNS channel multiplier. Then the modular reduction over GF(2^m) can be finally performed by PRNS addition with those component in (6).

The advantage of this approach is that, firstly, it reduces the complexity of the SRC conversion circuit as only half length of the intermediate product is converted, secondly, due to the partial conversion (lowers the probability of leaking the full information) and the adoption of the PRNS architecture, this approach is effective in preventing leaking information while performing the conversion and modular reduction. A detailed implementation of this method is shown in the next section.

In addition, the partial conversion method has found broad usage in overflow detection (for the PRNS error detection) and overflow prediction (used for the PRNS based AES) for the PRNS architecture, which will be further discussed in the following chapters.

3.4 GF(2^{163}) Multiplication using Trinomial based PRNS

This section introduces a improved design of implementing GF(2^m) multiplication using the PRNS. Irreducible trinomials are selected as the generating polynomials for the PRNS channels to enable conversion to-and-from PRNS to be implemented using simple XOR networks, thereby resulting in significant improvement in speed and area. The previously introduced PRNS modular reduction method over GF(2^m) is also adopted to achieve better performance.

3.4.1 Dynamic Range and Moduli Set

This design implements a GF(2^{163}) multiplier in PRNS (the same as previously introduced design in Chapter 3.2), aiming the application of the ECC using curve K-163. The large operands of such multiplier impose many design challenges.

To cover the whole dynamic range, four 84-degree irreducible trinomials are selected as the PRNS channels. This satisfies the dynamic range $d \times N \geq 2m$ equation, where $d = 84, N = 4, m = 163$. There are two main reasons why trinomials are chosen. Firstly, in Galois field multiplications, using trinomials achieves the lowest hardware complexity in modular reduction, especially when the trinomials are of the form $x^m + x^k + 1$, where $k \leq \frac{m}{2}$ [20]. This property of trinomials is also attractive when building the PRNS converter. Secondly, in the SRC algorithm, it can be seen that M_i, which is a constant value in the given PRNS, is the product of several channel-generating polynomials. To make the multiplying by M_i operation simple, it is necessary to require that M_i should have a smaller number of '1's and this can be best achieved by using trinomials, because they are the irreducible polynomials with the fewest '1's over binary field.

However, trade-offs are required between the channel length and the channel number for an optimized design. To cover the same dynamic range, shorter channel lengths require more channels, which may lead to consuming more hardware resources and to more complex converter design. In addition, irreducible trinomials only appear in certain degrees and the number of trinomials, which satisfy $k \leq \frac{m}{2}$, is even smaller. That is the reason why four 84-degree irreducible trinomials are selected for this design.

For the given PRNS, the GF(2^{163}) multiplication and the SRC conversion is then described as:

$$p(x) = A(x) \times B(b) \xrightarrow{PRNS} \{\langle a_1 \times b_1 \rangle_{m_1}, \langle a_2 \times b_2 \rangle_{m_2}, \langle a_3 \times b_3 \rangle_{m_3}, \langle a_4 \times b_4 \rangle_{m_4}\}$$

$$q(x) = p(x) \bmod f(x), \text{where } f(x) = x^{163} + x^7 + x^6 + x^3 + 1$$

$$p(x) = \sum_{i=1}^{4} (p_i(x) \cdot I_i(x) \bmod m_i(x)) \cdot M_i(x)$$

$$\text{Where } M_i(x) = \frac{M(x)}{m_i(x)} = m_1(x) \ldots \cdot m_{i-1}(x) \cdot m_{i+1}(x) \ldots \cdot m_4(x)$$

$$I_i = M_i^{-1}(x) \left(\bmod m_i(x)\right) \text{ for } i \in [1, 4]$$

The detailed information on the selection of $m_i(x)$'s and the value of the $M_i(x)$'s and $I_i(x)$'s are listed in the Appendix B.

3.4.2 Channel Multiplier Design

From the previous literature review, there are several approaches that can be adopted for implementing the PRNS channel multipliers, such as bit-serial architecture, bit-parallel architecture and digital serial/parallel architecture. Since the selected field length for PRNS channels is quite large, which is degree of 84, the bit-serial

architecture is adopted here in order to achieve the lowest hardware complexity. In Chapter 2, Figure 2-1 illustrates an MSB-first bit-serial multiplier over GF(2^4) generated by trinomial $x^4 + x + 1$ to demonstrate the architecture.

In addition, such multiplier is not only suitable for performing channel multiplication, but also for calculating $(p_i(x) \cdot I_i(x)\ mod\ m_i(x))$ in the SRC algorithm.

3.4.3 Multiplying by M_i Operation

Consider the following example where M_1 is written in polynomial form as:

$$M_1(x) = x^{252} + x^{181} + x^{179} + x^{177} + x^{168} + x^{108} + x^{106} + x^{104} + x^{84} + x^{33} + x^{24} + x^{22} + x^{20} + x^{13} + x^{11} + x^9 \quad (7)$$

(It is a product of all channels generating polynomials except the one for Channel 1. Those polynomials are $x^{84} + x^9 + 1$, $x^{84} + x^{11} + 1$, $x^{84} + x^{13} + 1$)

According to equation (5) and (6) in Chapter 3.3, since a partial conversion is performed to calculate the most significant 162 bits of the intermediate product, the component with the degree smaller than 84 can be ignored in (7), because they do not contribute to the final partial conversion result. So multiplying by M_1 can be done by multiplying by the following polynomial instead:

$$x^{252} + x^{181} + x^{179} + x^{177} + x^{168} + x^{108} + x^{106} + x^{104} + x^{84}$$

It is assumed that the multiplicand is $a(x)$ which is in a PB representation. The multiplication is as follows:

$$a(x) \cdot (x^{252} + x^{181} + x^{179} + x^{177} + x^{168} + x^{108} + x^{106} + x^{104} + x^{84})$$
$$= a(x) \cdot x^{252} + a(x) \cdot x^{181} + a(x) \cdot x^{179} + a(x) \cdot x^{177} + a(x) \cdot x^{168} + a(x) \cdot$$

$$x^{108} + a(x) \cdot x^{106} + a(x) \cdot x^{104} + a(x) \cdot x^{84} \quad (8)$$

It is simple to implement (8) by using an XOR network and routing $a(x)$ to the correct position as illustrated by Figure 3-4.

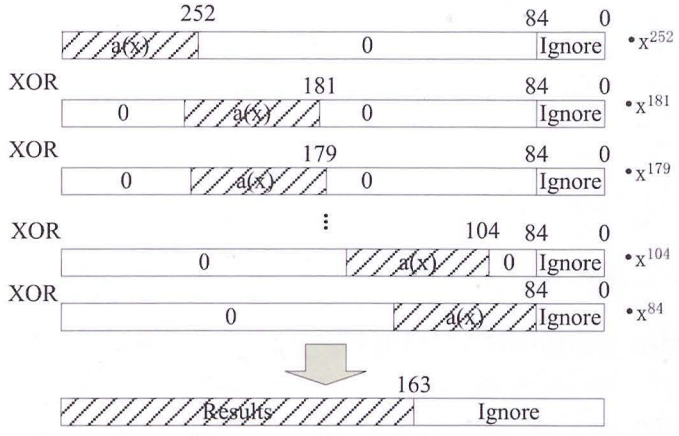

Figure 3-4: XOR Network for the Operation of Multiplying by M_1

3.4.4 GF(2^{163}) Modular Reduction and *to_PRNS* Converter

A bit-parallel modular reduction method is adopted to perform the field modular reduction to reduce the $(c_{2m-2}, c_{2m-3}, \cdots, c_m, 0, \cdots, 0)$ component to a degree smaller than m, the result is written in binary vector format as $(c'_{m-1}, \cdots, c'_1, c'_0)$, which implements the calculation of the first component in equation (6).

The *to_PRNS* converter is implemented by simple modular reduction using the selected trinomials. The detailed implementation approach can be found in [20].

Both modular reduction and conversion are implemented using a simple XOR network.

3.4.5 Architecture of the Proposed PRNS GF(2^{163}) Multiplier

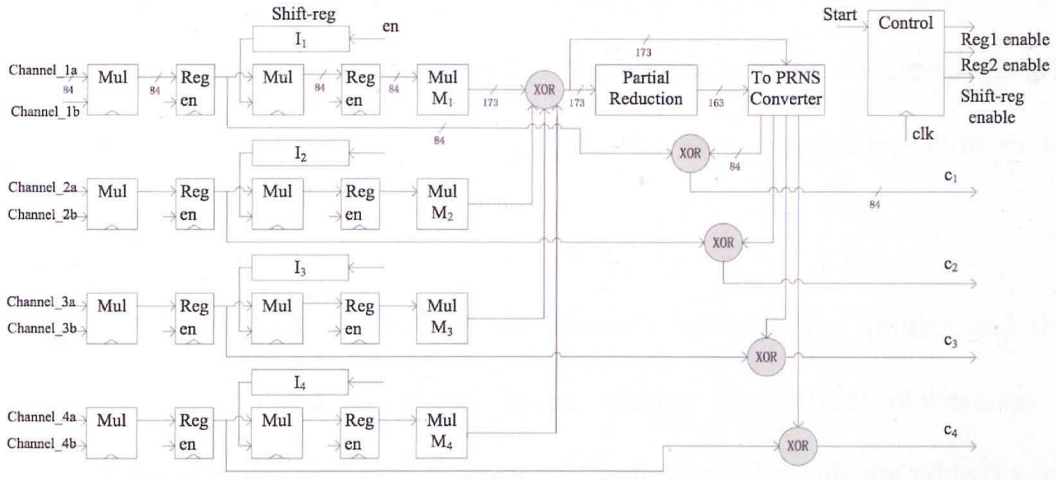

Figure 3-5: Architecture for the PRNS GF(2^{163}) Multiplier using Trinomials

Figure 3-5 shows the full architecture of the proposed PRNS multiplier over GF(2^{163}). It is assumed that the input and the output of the multiplier are all in PRNS representations.

The GF(2^{84}) multipliers on the left hand side perform PRNS channel modular multiplication, the one on the right performs part of SRC algorithm which is $(p_i(x) \cdot I_i(x) \ mod \ m_i(x))$ operation. $I_i(x)$'s are pre-calculated and stored into the shift registers. When there is a valid signal on the *Shift-reg* enable, the shift register starts to forward $I_i(x)$bitwisely to the second GF(2^{84}) multiplier acting as a bit-serial input.

The module *Mul* M_i, *Partial_Reduction* and *To_PRNS_Converter* are constructed using pure XORs. According to equation (6), the output of the *To_PRNS_Converter* is the PRNS representation of $(c_{2m-2}, c_{2m-3}, \cdots, c_m, c'_{m-1}, \cdots, c'_1, c'_0)$, the output of the first level of registers is the PRNS representation of $(c_{2m-2}, c_{2m-3}, \cdots, c_m, c_{m-1}, \cdots, c_1, c_0)$. The final product after GF(2^{163}) modular reduction is generated by a PRNS addition operation, which is implemented as

bitwise XORs.

It takes 168 clock cycles to finish a multiplication operation. This includes 83 clock cycles performing channel multiplication, another 83 clock cycles performing multiplication by $I_i(x)$ and 2 clock cycles on the data propagating through two registers.

As it can be seen from Figure 3-5, all channels are separate, similar and their operations are performed in parallel hence offering an inherent mechanism for masking, randomization and fault tolerance (if redundant channels are added) which could help improve protection against any potential side channel leakage or analysis [3].

3.4.6 Hardware Results and Comparisons

Xilinx Spartan 3-3s1500lfg320-4 FPGA is used for synthesis and implementation to enable a fair comparison. Table 3-3 shows the synthesis results of the proposed multipliers compared with the design that has been introduced in Chapter 3.2, which is the first reported implementation of a PRNS multiplier over binary fields.

From the results, the work that uses trinomials as the channel generating polynomials shows significant improvements both in hardware consumption and speed compared with our previous work. The figures indicate that this work consumes half and one third area consumption compared with the channel-serial and channel-parallel architecture respectively. The highest operating frequency is improved by over 30 times due to the reduction of the maximum combinational delay. The total delay is improved by 25 times over the channel-serial architecture and by 17 times over the channel-parallel architecture. These figures also show a 47 times' Time-Area Product

improvement over the channel-serial architecture, a 57 times' improvement over the channel-parallel architecture.

Table 3-3: Synthesis Results of the 4-Channel PRNS GF(2^{163}) Multiplier

	Channel-Serial	Channel-Parallel	This work
FF	1010	1350	1691
LUT	5274	8675	2588
Slices	2752	4625	1429
Frequency (MHz)	5.179	5.119	164.015
Cycles	130	93	168
Delay (ms)	$25.1*10^{-3}$	$18.2*10^{-3}$	$1.024*10^{-3}$
Time-Area Product (Slices*second)	$69*10^{-3}$	$84*10^{-3}$	$1.463*10^{-3}$

As mentioned in Chapter 3.4.1, using trinomials simplifies the modular reduction and the *To_PRNS* conversion operations. Furthermore, together with the partial conversion method, using trinomials breaks down the bottleneck in multiplying by $M_i(x)$ operation; hence it achieves higher speed and uses less resource.

Table 3-4: Comparisons with other GF(2^{163}) Implementation

Work implemented by [61]	Platform	Slices	Delay
Proposed by [58]	Virtex 2	5307	12.56μs
Proposed by [59]	Virtex 2	5409	13.37μs
Proposed by [60]	Virtex 2	5840	14.73μs
This work	Spartan 3	1429	1.024μs

Table 3-4 shows the comparisons with some other 163 bits parallel GF(2^m) multipliers. The figures indicate that this work shows great improvements both in area and speed. Though this multiplier is neither optimal on high speed nor on low area, it provides the potential to countermeasure side-channel-attacks as well as a feasible option to implement parallel architectures.

3.5 Functional Testing

The correctness of the proposed design in this chapter has been verified using ModelSim based test bench in VHDL. Since there are no direct testing vectors for such PRNS architecture, extra pre-tested error-free testing circuits are built to help with the testing process. The testing circuit setup is shown in Figure 3-6:

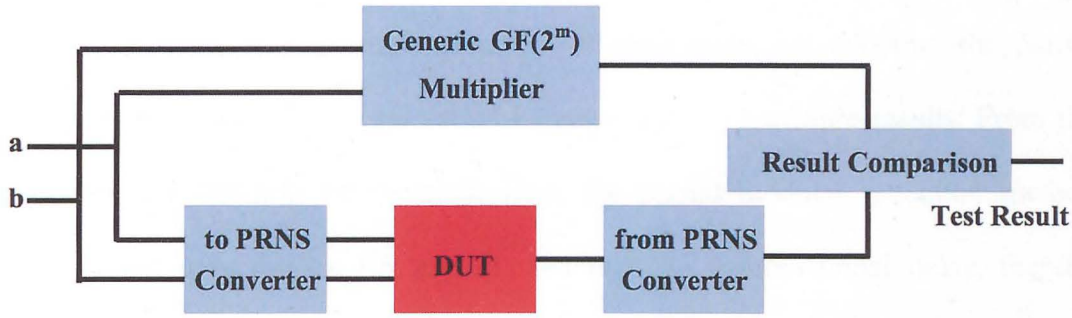

Figure 3-6: Testing Circuits for the PRNS GF(2^m) Multiplier

The testing process is simple. There are two branches of the testing circuits, the first one calculates a GF(2^m) multiplication using normal polynomial representation, the other branch calculates the multiplication using the proposed PRNS multiplier using the same operands which are previously converted to PRNS format through the *to_PRNS* Converter. When the calculation is done, the results from the two branches are compared to check if they are identical to verify the design. The input testing vectors are chosen from random GF(2^m) elements, while the referencing testing results are generated on the fly from the Generic GF(2^m) Multiplier.

3.6 Conclusions

This chapter described the design of a novel PRNS based GF(2^m) multiplier. Three different architectures together with the implementation results are given. From the synthesis results, it is noticed that the conversion circuit causes main overhead in such PRNS multiplier. To overcome this obstacle, a novel conversion and modular reduction method is introduced to the PRNS architecture, namely partial modular reduction method. A new implementation of such multiplier adopting the partial modular reduction method is presented together with its hardware results. From the comparison of different PRNS multipliers, the partial modular reduction method enables great reduction in the use of area and the combinational delay, thereby improves the performance, which makes such PRNS multiplier feasible for the mentioned application ECC using curve K-163.

The next chapter will look into the error detection and fault tolerant property that is brought by the PRNS architecture. The error detection and fault tolerance GF(2^m) multiplier designs are also presented.

Chapter 4
PRNS for Error Detection and Fault Tolerance

4.1 Introduction

The increasing use of Galois fields GF(2^m) arithmetic in cryptographic applications requires the implementation of GF(2^m) circuits to have higher level of reliability. Furthermore, in recent years, a new approach of attacking a cryptographic system named fault attacks uses the leaking information generated by the system's faulty operations to obtain the analytical results of the secret information [62]. Preventing faulty operations is becoming an important issue in cryptographic applications design.

The ECC and AES are both shown to be vulnerable to fault attacks in literatures. Works in [63, 64] present a few fault attack schemes against the public key scheme ECC. In the AES, prior work has shown that even a single transient error occurring during the AES round operations will very likely result in a large number of errors in the final data [68]. In addition, a few attack scenarios have shown that the AES is quite vulnerable to fault attacks as well [68, 69, 70, 71, 72]. Hence it is necessary to provide error detection mechanisms to the cryptography designs to achieve higher level of reliability and security.

Since GF(2^m) multiplication is the crux operation in ECC, some work have been done by researchers to incorporate error detection or correction to the multiplier over GF(2^m). Works in [65, 66] use parity-based approaches to achieve error detection and

[67] uses an alternative method based on a re-computing with shifted operands (RESO) method.

In this chapter, a new approach using the PRNS architecture to implement the $GF(2^m)$ multiplier with error detection capability is presented. In this approach, error detection in the multiplication over $GF(2^m)$ is achieved by using PRNS representation and extended bases (or called the Redundant PRNS, RPRNS). The background theories and mathematic proof is given in the first part of this chapter, followed by a detailed FPGA implementation of a $GF(2^8)$ error detection multiplier as a demonstration of the proposed error detection method. Furthermore, based on the error detection capability that is provided by the RPRNS, concurrent error correction (or known as the fault tolerance) is achieved by adding two or more redundant channels. An example of such fault tolerance $GF(2^{163})$ multiplier design (for the ECC scheme) together with its implementation results are also presented. This chapter is concluded with the overhead and error coverage analysis.

4.2 The RPRNS Based Error Detection

The PRNS over GF(2^m) based error detection is similar to RNS error detection over integers, the proof of which is given in [73]: by adding a redundant polynomial residue channel, the whole representation range is then divided into two intervals: the legitimate range and illegitimate range [74]. Any error in a single channel can be detected if its conversion result belongs to the illegitimate range.

The Redundant PRNS (RPRNS) is defined using a normal PRNS that uses the irreducible polynomial set $m_1(x), m_2(x), \cdots, m_N(x)$ and the sum of its degree satisfies the $\sum_{i=1}^{N} d_i \geq 2m$ equation, which covers the dynamic range of the intermediate product, added with one additional polynomial moduli channel using the polynomial $m_{N+1}(x)$ with the degree $d_{N+1} \geq d_i$ for $i \in [1, N]$.

The product polynomial $M(x) = \prod_{i=1}^{N} m_i(x)$ represents the legitimate range, where the possible highest degree is denoted as $D = \sum_{i=1}^{N} d_i$. After adding the module m_{N+1}, the whole representation range is then described as $M'(x) = \prod_{i=1}^{N+1} m_i(x)$, where the possible highest degree of $M'(x)$ is denoted as $D' = \sum_{i=1}^{N+1} d_i$. Those polynomials with the highest degree greater or equal to D and smaller than D' constitutes the illegitimate range, which indicates an error. The proof is given below:

Assuming the intermediary product of two arbitrary GF(2^m) elements' multiplication X is represented in redundant RPRNS form as $\{x_1, x_2, \cdots, x_N, x_{N+1}\}$, which belongs to the legitimate range with the degree no higher than D, when a single channel error occurs in the ith channel while multiplying, the result yields a faulty RPRNS representation of $\breve{X}$ as $\{x_1, \cdots, \breve{x_i}, \cdots, x_{N+1}\}$. The $\breve{X}$ can be represented using its correct value X added by the error value $\breve{E}$ as:

$\check{X} = X + \check{E}$ or in PRNS format as:

$$\{x_1, \cdots, \check{x}_i, \cdots, x_{N+1}\} = \{x_1, \cdots, x_i, \cdots, x_{N+1}\} + \{0, \cdots, \check{e}_i, \cdots, 0\}$$

Since X represent the intermediary product of two arbitrary elements' multiplication over GF(2^m), its degree will not exceed *2m-2*, in another word, X belongs to the legitimate range. By converting $\check{E}(x)$'s RPRNS representation back to weighted polynomial representation using the SRC algorithm, it yields:

$$\check{E}(x) = (\check{e}_i(x) \cdot I_i'(x) mod\ m_i(x)) \cdot M_i'(x)$$

$$\text{Where } M_i'(x) = \frac{M'(x)}{m_i(x)} = m_1 \ldots \cdot m_{i-1} \cdot m_{i+1} \ldots \cdot m_N \cdot m_{N+1}$$

$$I_i'(x) = M_i'^{-1}(x)(mod\ m_i(x))$$

The highest degree of $M_i'(x)$ is $\sum_{i=1}^{N+1} d_i - d_i$ and the degree of $(\check{e}_i(x) \cdot I_i'(x)\ mod\ m_i(x))$ is from 0 to $d_i - 1$, so the possible highest degree of $\check{E}(x)$, which is denoted as $D_{\check{E}}$ ranges from $(\sum_{i=1}^{N+1} d_i - d_i)$ to $(\sum_{i=1}^{N+1} d_i - 1)$.

For $i \in (1 \text{ to } N)$, (error occurs in the normal RPNS channal) then

$$\sum_{i=1}^{N+1} d_i - d_i \leq D_{\check{E}} \leq \sum_{i=1}^{N+1} d_i - 1$$

$$\sum_{i=1}^{N} d_i + d_{N+1} - d_i \leq D_{\check{E}} \leq \sum_{i=1}^{N+1} d_i - 1$$

Since $d_{N+1} \geq d_i > 1$ for $i \in (1 \text{ to } N)$,

$$D = \sum_{i=1}^{N} d_i \leq D_{\check{E}} < \sum_{i=1}^{N+1} d_i = D'$$

$$D \leq D_{\check{E}} < D'$$

For $i = N + 1$, (error occurs in the redundant channel) then

$$\sum_{i=1}^{N+1} d_i - d_{N+1} \leq D_{\check{E}} \leq \sum_{i=1}^{N+1} d_i - 1$$

$$D = \sum_{i=1}^{N} d_i \leq D_{\check{E}} < \sum_{i=1}^{N+1} d_i = D'$$

$$D \leq D_{\check{E}} < D'$$

All possible cases indicate that the error vector $\check{E}$ belongs to the illegitimate range, hence the faulty intermediate product $\check{X} = X + \check{E}$ belongs to the illegitimate range.

An example to demonstrate the proposed RPRNS based error detection using a GF(2^8) multiplier is given in the following section.

4.3 GF(2^m) Multiplier using RPRNS Based Error Detection

4.3.1 Example on RPRNS Based Error Detection

In this section, an example is illustrated by using the multiplication over GF(2^8) generated by the irreducible polynomial $f(x) = x^8 + x^4 + x^3 + x + 1$, which is the officially defined binary field for the AES algorithms. The binary vector form of Polynomial Representation for GF elements is adopted to simplify the representation.

To construct the redundant PRNS, the following channel generating polynomials are selected, such that $\sum_{i=1}^{N} d_i \geq 2m$ and $d_{N+1} \geq d_i$ for $i = 1$ to 3:

$$m_1(x) = x^6 + x + 1\ (1000011)$$
$$m_2(x) = x^6 + x^5 + 1\ (1100001)$$
$$m_3(x) = x^6 + x^3 + 1\ (1001001)$$
$$m_4(x) = x^6 + x^4 + x^2 + x + 1\ (1010111)$$

The constant value of M_i and I_i is pre-calculated and listed as:

$$M_1 = 1110011100100001111$$
$$M_2 = 1011111001010110001$$
$$M_3 = 1111111110001011001$$
$$M_4 = 1101110011100111011$$
$$I_1 = M_1^{-1}(mod\ m_1) = 100011$$
$$I_2 = M_2^{-1}(mod\ m_2) = 000110$$
$$I_3 = M_3^{-1}(mod\ m_3) = 100000$$
$$I_4 = M_4^{-1}(mod\ m_4) = 000010$$

For example, A and B are elements in the defined GF(2^8) field, which is (10011101) and (01100111) respectively. A and B are written in RPRNS format as:

$$A_i = A \bmod m_i, \text{for } i = 1 \text{ to } 4$$

$$A_1 = (10011101) \bmod (1000011) = 011011$$

$$A_2 = (10011101) \bmod (1100001) = 111110$$

$$A_3 = (10011101) \bmod (1001001) = 001111$$

$$A_4 = (10011101) \bmod (1010111) = 110011$$

Similarly, B's RPRNS representation can also be obtained:

$$B_1 = 100100, B_2 = 000110$$

$$B_3 = 101110, B_4 = 110000$$

Then the multiplication is performed in RPRNS as, Pdt is used to present the correct product of the multiplication, which is also in the RPRNS form:

$$Pdt = P \times Q = \{\langle A_1 \times B_1 \rangle_{m_1}, \langle A_2 \times B_2 \rangle_{m_2}, \langle A_3 \times B_3 \rangle_{m_3}, \langle A_4 \times B_4 \rangle_{m_4}\}$$
$$= \{\langle 011000 \rangle, \langle 100111 \rangle, \langle 001100 \rangle, \langle 100000 \rangle\}$$

If Pdt is converted back to weighted polynomial representation, the result will stay in the legitimate range. Let's verify it:

$$Pdt = \sum_{i=1}^{4} (Pdt_i \cdot I_i \bmod m_i) \cdot M_i$$

$$= [(011000 \times 100011) \bmod (1000011)] \times (1110011100100001111)$$
$$+ [(100111 \times 100011) \bmod (1100001)] \times (1011111001010110001)$$
$$+ [(001100 \times 100011) \bmod (1001001)] \times (1111111110001011001)$$
$$+ [(100000 \times 100011) \bmod (1010111)] \times (1101110011100111011)$$
$$= 101100011001110101010100 \text{ XOR } 010111110010101100010000$$
$$\text{XOR } 100100000100011101100110 \text{ XOR } 011111101100011000010001$$
$$= (000000000)011011100110011$$

The most significant 9 bits of the intermediate product indicates whether this product stays in the legitimate range. If any error, either stuck at '1' or '0' error or multiple errors, occurs in one channel, there will be at least one '1' in the most significant 9 bits of the product to indicate an error occurring.

For example, there is a single bit error that occurs in the third channel, which may be either caused by natural reasons such as abnormal temperature, power supply variations, electromagnetic interference, or by an adversary fault injection. Its value changes from 001100 to 001101. Then the faulty intermediate product Pdt' is calculated as:

$$Pdt' = \sum_{i=1}^{4} (Pdt'_i \cdot I_i \bmod m_i) \cdot M_i$$

$$= [(011000 \times 100011) \bmod (1000011)] \times (1110011100100001111)$$
$$+ [(100111 \times 100011) \bmod (1100001)] \times (1011111001010110001)$$
$$+ [(\mathbf{001101} \times 100011) \bmod (1001001)] \times (1111111110001011001)$$
$$+ [(100000 \times 100011) \bmod (1010111)] \times (1101110011100111011)$$
$$= 101100011001110101010100 \text{ XOR } 010111110010101100010000$$
$$\text{XOR } 011011111100110001000110 \text{ XOR } 011111101100011000010$$
$$= (111111111)011110000010011$$

The above bold digits indicate how the faulty channel results will influence the conversion result. Those '1's in bracket shows the error occurs.

Let's see another example, also in Channel 3 where multiple bits error occurs. The value from Channel 3 changes from 001100 to 110101, then the faulty intermediate product Pdt'' is computed as:

$$Pdt'' = \sum_{i=1}^{4} (Pdt''_i \cdot I_i \bmod m_i) \cdot M_i$$

$$= [(011000 \times 100011) \bmod (1000011)] \times (1110011100100001111)$$
$$+ [(100111 \times 100011) \bmod (1100001)] \times (1011111001010110001)$$
$$+ [(110101 \times 100011) \bmod (1001001)] \times (1111111110001011001)$$
$$+ [(100000 \times 100011) \bmod (1010111)] \times (1101110011100111011)$$
$$= 101100011001110101010100 \text{ XOR } 010111110010101100010000$$
$$\text{XOR } 100001111011001011101001 \text{ XOR } 011111101100011000010001$$
$$= (000101111)100001010111100$$

As it can be seen from the result, there are still '1's in the most significant 9 bits, by which an error can be determined.

Two examples indicate that, the PRNS based error detection scheme is capable of detecting single bit errors and multiple bits errors that occurs in one channel.

4.3.2 Implementation of GF(2^8) Error Detection Multiplier Using RPRNS

In this section, a detailed implementation of the GF(2^8) error detection multiplier using the irreducible polynomial $f(x) = x^8 + x^4 + x^3 + x + 1$, which is the officially defined binary field for the AES algorithm, is presented to demonstrate the proposed PRNS architecture. This multiplier can be used to provide the AES designs with error detection capability.

In order to cover the whole dynamic range, the equations $\sum_{i=1}^{N} d_i \geq 2\times 8 = 16$ and $d_{N+1} \geq d_i$ need to be satisified. Furthermore, taking the complexity of the channel multipliers into account, using trinomials achieves the lowest hardware complexity in modular reduction [20] and shows further advantages in building the PRNS converter (Chapter 3.4). As a result, the following irreducible polynomials are chosen as the moduli set:

$$m_1(x) = x^6 + x + 1\ (1000011)$$

$$m_2(x) = x^6 + x^5 + 1\ (1100001)$$

$$m_3(x) = x^6 + x^3 + 1\ (1001001)$$

$$m_4(x) = x^6 + x^4 + x^2 + x + 1\ (1010111)\ \text{(redundant moduli)}$$

Then the multiplication is denoted as:

$$p(x) = A(x) \times B(b) \xrightarrow{RPRNS} \{\langle a_1 \times b_1 \rangle_{m_1}, \langle a_2 \times b_2 \rangle_{m_2}, \langle a_3 \times b_3 \rangle_{m_3}, \langle a_4 \times b_4 \rangle_{m_4}\}$$

For the channel multiplier, since the field length for each channel is short, bit-parallel architecture introduced in [20] is adopted to implement the channel multiplier.

Taking Channel 1, which is generated by $m_1(x) = x^6 + x + 1$, as an example:

Assuming $a(x)$ and $b(x)$ are elements over GF(2^6), $c(x)$ is the product, the multiplication performs as:

$$c(x) = [a(x) \times b(x)] \bmod m_1(x) = (c_{10}x^{10} + c_9x^9 + c_8x^8 + c_7x^7 + c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0) \bmod (x^6 + x + 1)$$

$$= (c_5 + c_{10})x^5 + (c_c + c_9 + c_{10})x^4 + (c_3 + c_8 + c_9)x^3 + (c_2 + c_7 + c_8)x^2 + (c_1 + c_6 + c_7)x + (c_0 + c_6)$$

Where $c_i = 0$ or 1 for $i = 0$ to 10, which is the coefficient of the polynomial representation of the intermediary product.

The conversion follows the SRC algorithm that is introduced in Chapter 2.4.2 as:

$$p(x) = \sum_{i=1}^{4} (p_i(x) \cdot I_i(x) \bmod m_i(x)) \cdot M_i(x)$$

This converter performs the conversion from PRNS representation to normal weighted polynomial representation, which is also needed for the error detection.

From the above equation, there are three main operations to perform the conversion: the modular multiplication with pre-calculated constant $I_i(x)$, the normal multiplication with $M_i(x)$ that is also a constant value generated by the chosen moduli

set and the final sum operation. For the detailed information of the constant values, please refer to the previous section Chapter 4.3.1.

The implementation of the modular multiplication with $I_i(x)$ applies the same technique that presented in [56]. Taking Channel 2, which is generated by $m_2(x)$ as an example: the I_2 is $x^2 + x$ in its PB representation, the modular multiplication is demonstrated as follows: assuming the result is $y(x)$,

$$\begin{aligned} y(x) &= [a(x) \times I_2(x)] \bmod m_2(x) \\ &= (a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0) \cdot (x^2 + x) \bmod (x^6 + x^5 + 1) \\ &= [a_5x^7 + (a_4 + a_5)x^6 + (a_3 + a_4)x^5 + (a_2 + a_3)x^4 + (a_1 + a_2)x^3 \\ &\qquad + (a_0 + a_1)x^2 + a_0x] \bmod (x^6 + x^5 + 1) \\ &= a_3x^5 + (a_2 + a_3)x^4 + (a_1 + a_2)x^3 + (a_0 + a_1)x^2 + (a_0 + a_5)x + a_4 \end{aligned}$$

The above calculation can be implemented using simple XOR network as showed in Figure 4-1.

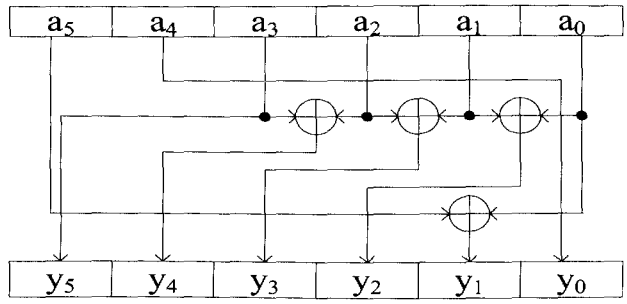

Figure 4-1: Implementation Example of Multiplying by I_i Operation

The implementation of the multiplying by $M_i(x)$ operation adopts the same method that introduced in the previous chapter (see Chapter 3.4.3). The final sum operation of the SRC algorithm is implemented using simple bitwise XOR operation.

To use the proposed RPRNS error detection method, firstly, the operands of the multiplication need to be converted to the RPRNS format according to the selected channel generating polynomials. The *to_PRNS* converter converts the original

weighted polynomial representation over GF(2^8) to the RPRNS representation by using four GF(2^6) elements which are the residues of $m_1(x), m_2(x), m_3(x), m_4(x)$ respectively. The conversion is performed as the modular reduction operation over GF(2^6). The same modular reduction method used for designing the channel multiplier can be applied for the converter.

For example, the original GF(2^8) representation is written as $o(x)$, the converted RPRNS representation for Channel 3 using m_3 ($x^6 + x^3 + 1$) is $z(x)$

Then the conversion to the RPRNS is done as:

$$z(x) = o(x) \bmod m_3(x)$$
$$= o_7x^7 + o_6x^6 + o_5x^5 + o_4x^4 + o_3x^3 + o_2x^2 + o_1x + o_0 \bmod (x^6 + x^3 + 1)$$
$$= o_5x^5 + (o_7 + o_4)x^4 + (o_6 + o_3)x^3 + o_2x^2 + (o_7 + o_1)x + (o_6 + o_0)$$

This operation can be implemented using simple XOR network, similar architecture as shown in Figure 4-1. The same design methodology is applied to other channels as well and all the channel multiplication are performed in parallel.

To ensure the multiplication is closed, a modular reduction operation using $f(x) = x^8 + x^4 + x^3 + x + 1$ is required.

According to [56], the modular reduction can be implemented as:

Assuming the intermediate product is denoted as $p(x)$ which is with the highest degree $2\times8 - 2 = 14$ and the results after modular reduction as $z(x)$, let's use their polynomial coefficients' binary vector form to simplify the expression, where $p(x) = (p_{14}p_{13} \cdots p_1p_0), z(x) = (z_7z_6 \ldots z_1z_0)$, then:

$z_7 = p_7 + p_{11} + p_{12} + p_{14}$, $\quad z_6 = p_6 + p_{10} + p_{11} + p_{13}$,

$$z_5 = p_5 + p_9 + p_{10} + p_{12}, \qquad z_4 = p_4 + p_8 + p_9 + p_{11} + p_{14},$$

$$z_3 = p_3 + p_8 + p_{10} + p_{11} + p_{12} + p_{13} + p_{14},$$

$$z_2 = p_2 + p_9 + p_{10} + p_{13},$$

$$z_1 = p_1 + p_8 + p_9 + p_{12} + p_{14}, \qquad z_0 = p_0 + p_8 + p_{12} + p_{13}.$$

Figure 4-2 shows the full architecture of the proposed error detection PRNS multiplier $c = ab \bmod f(x)$ over GF(2^8). The validity of this multiplier is based on the following assumptions: the conversion circuit, final modular reduction circuit and the error detection module can be made fault free.

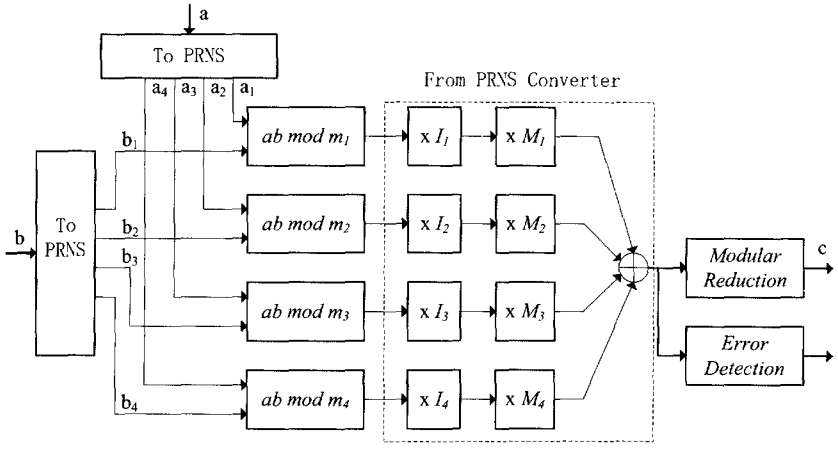

Figure 4-2: Architecture of the GF(2^8) RPRNS Error Detection Multiplier

The error detection module checks if the highest degree of the converted weighted polynomial is in the illegitimate range. If all the inputs are '0', which indicates the degree of the polynomial is smaller than those in illegitimate range, the output will give a 'no error' signal; however, if a '1' (or more than one '1') appears in the input, which means the highest degree is in the illegitimate range, the error detection module will yield a 'error' signal. A 9-bit OR gate is used to achieve above function in this design.

As the multiplication is over small Galois fields, the proposed architecture is

constructed using pure combinational logic circuits. As it can be seen from Figure 4-2, all channels are separate, similar and their operations are performed in parallel hence offering an inherent mechanism for masking, randomization and which could help improve protection against any potential side channel leakage or analysis.

4.3.3 Implementation Results

Xilinx Spartan 3-3s1500lfg320-4 FPGA is used for synthesis and implementation for the propose error detection multiplier.

Table 4-1 shows the synthesis results of the proposed PRNS error detection multiplier. To the authors' knowledge, there are not any hardware synthesis results for such PRNS architecture error detecting GF(2^m) multipliers to compare with. The comparison with a standard bit parallel GF(2^m) multiplier and a PRNS GF(2^m) multiplier without error detection is made to analyse the overhead of the error detection functionality.

Table 4-1: GF(2^8) Multiplier Synthesis Results

	Slices	LUTs	Max Combinational Delay
i.	**141**	**246**	**28.766ns**
ii.	**99**	**173**	**28.851ns**
iii.	**31**	**55**	**16.991ns**

i. The implementation of the GF(2^8) error detecting multiplier using RPRNS

ii. The implementation of a GF(2^8) multiplier using three 6-bit PRNS channels, without error detecting capability

iii. The implementation of a bit-parallel GF(2^8) multiplier using the method that is introduced in [56]

As it can be seen from the above table, as expected, the PRNS architecture has higher level of complexity than a standard GF(2^m) multiplier that results in larger area consumption and longer operating delay. However, due to the nature of independence between PRNS channels and scope for randomisation, this architecture has much more scope for improving side-channel resistance in cryptosystems. In addition, by

introducing redundant channels, error detection can be achieved and if more redundant channels are introduced within the PRNS architecture, fault tolerant design can also be implemented with a cost of a larger overhead. Compared with a PRNS multiplier, there is a 40% overhead on area to achieve error detection caused by introducing the redundant channel and the increased complexity of the converter.

Table 4-2: GF(2^{163}) Multiplier Synthesis Results

	Slices	Clock Cycle	Max Frequency (MHz)
i.	**1429**	**168**	**164.015**
ii.	**3173**	**168**	**164.015**
iii.	**2245**	**254**	**163.639**

i. The implementation of GF(2^{163}) PRNS multiplier using four 84-bit channels (Chapter 3.4)

ii. The implementation of a GF(2^{163}) RPRNS multiplier using five 84-bit channels, with error detecting capability, using the proposed method

iii. The implementation of a GF(2^{163}) RPRNS multiplier using four 127-bit channels, with error detecting capability, using the proposed method

Table 4-2 shows the synthesis results of the GF(2^{163}) PRNS multipliers which are suitable for ECC applications. Along with the increasing number of channels, the area consumption increases dramatically. By looking at the design **i.** and **ii.**, the introducing of a redundant channel increases the parallelism of the design and the complexity of the conversion circuit which causes over 100% overhead in hardware. To improve the performance, design **iii.** uses a smaller number of channels, but, in order to cover the same dynamic range, it increases the channel length at the same time. Synthesis results show implementation **iii.** has much smaller overhead in area compared with implementation **ii.**. It is to say, the overhead that is needed to achieve error detection is adjustable according to the number of channels and channel complexity such as channel length and channel generating polynomials, which offers the designer great flexibility to construct the error detecting multiplier.

4.3.4 Error Coverage Analysis

The correctness of the proposed error detecting strategy is based on the assumptions that the conversion and modular reduction unit (as shown in Figure 4-2) are implemented in a secure environment – either hardware or software, where no error or fault can be injected.

The proposed error-detecting scheme is capable of detecting 100% single bit errors and 100% channel errors, where error occurs only in one channel, up to X-bit multiple errors where X is the channel field length. If multiple faults occur across different channels, the probability of detecting the error by this scheme is (only those error patterns that stay in the legitimate range will be missed) calculated as:

$$1 - \frac{2^{\sum_{i=1}^{N} d_i}}{2^{\sum_{i=1}^{N+1} d_i}} = \frac{2^{\sum_{i=1}^{N+1} d_i} - 2^{\sum_{i=1}^{N} d_i}}{2^{\sum_{i=1}^{N+1} d_i}}$$

Where d_i indicates the degree of the channel generating polynomial, N is the number of channels of the RPRNS.

To simplify the expression, both the numerator and denominator of the above equation divide $2^{\sum_{i=1}^{N} d_i}$, it yields:

$$\text{Error Detection Probability} = \frac{2^{d_{N+1}} - 1}{2^{d_{N+1}}}$$

Where d_{N+1} is the degree of the redundant channel.

According to the above equation, the error detection probability of cross channel multiple errors for the proposed design is calculated as shown in Table 4-3:

Table 4-3: Error Coverage for the Proposed Designs of GF Multiplier

Designs	Error Detection Probability
i. GF(2^8) multiplier using four 6-bit PRNS channels	$\frac{2^6-1}{2^6} = 98.4375\%$
ii. GF(2^{163}) multiplier using five 84-bit PRNS channels	$\frac{2^{84}-1}{2^{84}} \approx 100\%$
iii. GF(2^{163}) multiplier using four 127-bit channels	$\frac{2^{127}-1}{2^{127}} \approx 100\%$

As it can be seen from Table 4-3, the probability of undetected errors decreases exponentially when the channel length increases. The probability of undetected error patterns in design **ii.** is $\frac{1}{2^{84}} \approx 5.16\times10^{-26}$ while the probability in design **iii.** is $\frac{1}{2^{127}} \approx 5.88\times10^{-39}$, hence the error detection probability in these two designs tends towards 100%.

4.4 The RPRNS Based Fault Tolerance

The fault tolerant capability (or error correction capability) that is provided by the PRNS is based on the data independency of each channel and the reduplicative residue representation of the original data. In other words, the original data is represented reduplicatively using different PRNS channel combinations, if error occurs in one channel, the PRNS architecture will ignore the data from the faulty channel, use the error free residue representation of the original data to carry on with other operations. To perform the error correction, firstly, it is needed to locate the channel in which the error occurs; then, convert the PRNS representation back to normal representation bypassing the faulty channel, where the rest channels can still cover the dynamic range; in the end, convert the error free normal representation back to PRNS representation and use the correct data to replace the faulty channel data.

To achieve fault tolerance, it requires at least two additional moduli with respect to the normal PRNS representation, one additional channel is to detect an error, the other additional is used to locate in which channel the error occurs; hence the moduli set for such RPRNS $GF(2^m)$ multiplier can be denoted as $m_1(x), m_2(x), \cdots, m_N(x), m_{N+1}(x), m_{N+2}(x)$, where the sum of its degree satisfies the $\sum_{i=1}^{N} d_i \geq 2m$ equation and $d_j \geq d_i$ for $i \in [1, N]$, $j \in [N+1, N+2]$. In such RPRNS, arbitrary $N+1$ channels are capable of covering the entire dynamic range of the multiplication and providing error detection, so that $N+2$ sets of SRC conversion are required to locate the erroneous channel, where each SRC conversion is constructed using moduli $m_1(x), m_2(x), \cdots, m_{i-1}(x), m_{i+1}(x), \cdots, m_{N+2}(x)$, where i indicates the SRC set number. Figure 4-3 demonstrates a fault tolerant RPRNS architecture with 3 normal

PRNS channels and 2 redundant channels:

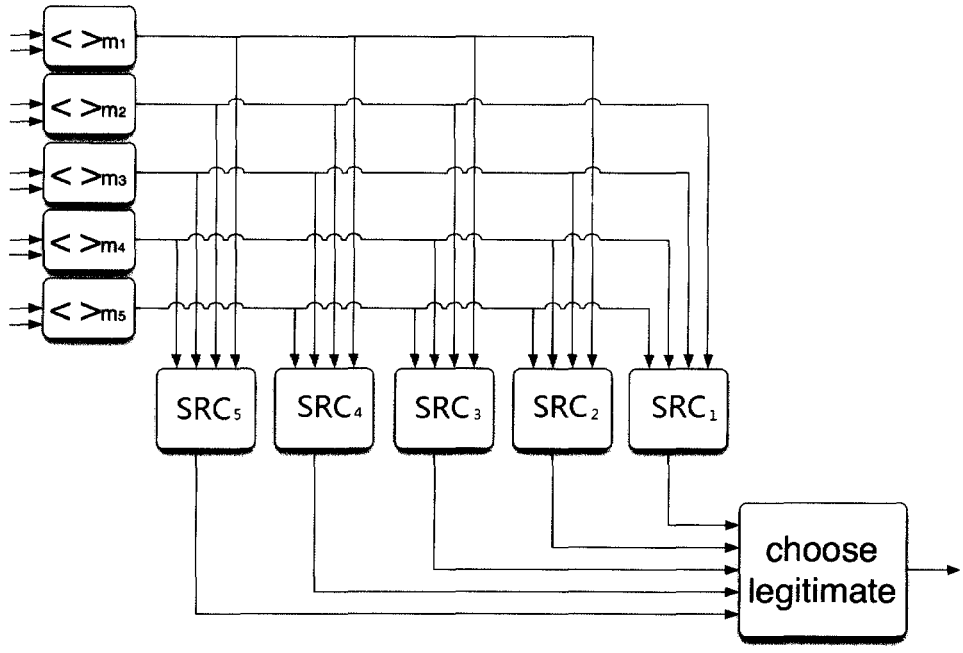

Figure 4-3: Architecture of RPRNS Based Fault Tolerance

For example, if an error occurs in a channel, say Channel 3, after the SRC conversions, the results from the SRC blocks which contains Channel 3, will fall into the illegitimate range (see Chapter 4.3), which indicates a channel error, except the result from the SRC_3 block because it bypasses the faulty channel. The choose_legitimate block will compare the results and choose the error free result, which is in the legitimate range, for further operations. The choose_legitimate block can be constructed using AND gates to detect the overflow in the illegitimate range and a simple multiplexer to forward the error free result.

4.5 GF(2^m) Multiplier using RPRNS Based Fault Tolerance

4.5.1 Implementation of Fault Tolerant GF(2^{163}) Multiplier

The implementation of a fault tolerant GF(2^{163}) multiplier using five 127-bit channel RPRNS is presented in this section. The fault tolerant design is based on the four 127-bit channel error detection design that has been presented in Chapter 4.3.3. By adding another redundant channel to the error detection design, the new multiplier is capable of providing fault tolerance to against internal errors and fault injections. The detailed information of the selected channel generating polynomials and the constant values for the SRC conversion are listed in Appendix C.

The proposed architecture of the fault tolerant multiplier is shown in Figure 4-4:

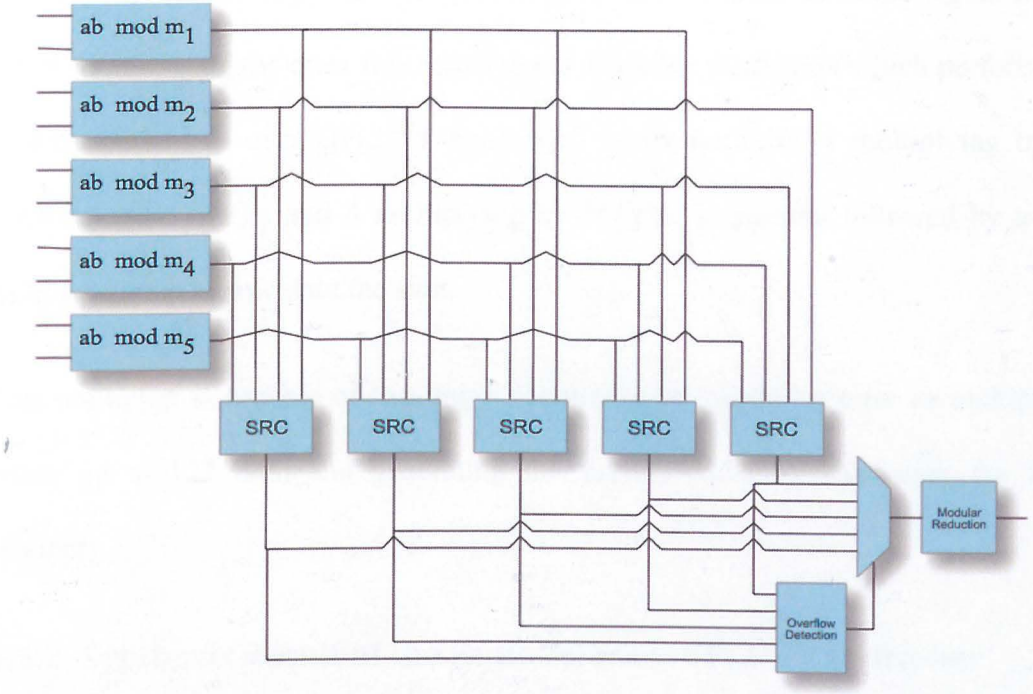

Figure 4-4: Architecture of RPRNS Based Fault Tolerance GF(2^{163}) Multiplier

Each SRC block is constructed as shown in Figure 4-5:

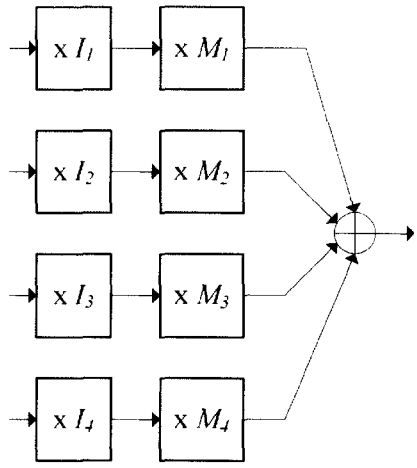

Figure 4-5: Architecture of the SRC Block

In Figure 4-4 each channel arithmetic block performs the channel multiplication over GF(2^{127}), the SRC blocks convers the selected channels back to the normal polynomial representation, the *Overflow_Detection* block checks if the results from the SRCs are in the legitimate range and generates the right selection signal for the MUX to propagate the error free result to the *Modular_Reduction* which performs the modular reduction over GF(2^{163}). Each SRC block performs 4 multiplying by the constant value $I_i(x)$'s and 4 multiplying by $M_i(x)$'s in parallel followed by a final XOR operation to calculate the sum.

This multiplier is capable of locating a channel error (signal bit error or multiple bit errors up to 127 bits) and generating the correct result by bypassing the faulty channel.

4.5.2 Synthesis Result of the Fault Tolerant GF(2^{163}) Multiplier

The FPGA synthesis result of the designed multiplier together with the comparison with the normal PRNS multiplier and the RPRNS based error detection architecture

are presented in Table 4-4:

Table 4-4: Synthesis Result of the Fault Tolerant GF(2^{163}) Multiplier

	Slices	Clock Cycle	Max Frequency (MHz)
iv.	**1429**	**168**	**164.015**
v.	**3173**	**168**	**164.015**
vi.	**2245**	**254**	**163.639**
vii.	**10307**	**254**	**163.345**

iv. The implementation of GF(2^{163}) PRNS multiplier using four 84-bit channels (Chapter 3.4)

v. The implementation of a error detection GF(2^{163}) RPRNS multiplier using five 84-bit channels

vi. The implementation of a error detection GF(2^{163}) RPRNS multiplier using four 127-bit channels

vii. The implementation of the proposed fault tolerant GF(2^{163}) multiplier using five 127-bit RPRNS channels

As it can be seen from the above table, the hardware overhead dramatically increases in the fault tolerant design (almost 400% overhead compared with the error detection design). The main overhead is related to the implementation of the SRC conversion blocks. For a RPRNS with $N + 2$ channels, it needs $N + 2$ SRC blocks, each of which is composed by:

- $N + 1$ modulo $m_i(x)$ constant multipliers to calculate $a(x) \cdot I_i(x) \bmod m_i(x)$
- $N + 1$ constant multiplier for M_i
- N XOR operations to calculates the sum

Therefore, the overall overhead due to the SRC conversions grows quadratically with the number of channels. Due to using the same architecture, the proposed designs achieve similar level of the maximum operating frequency.

4.6 Conclusions

This chapter introduced the error detection and error correction capability that is provided by the PRNS architecture. The mathematic proof of the error detection capability is given in the first place, followed by a detailed example and implementation of an 8-bit multiplier. Then the implementation result of such error detection multiplier for GF(2^{163}) is also presented together with the error coverage analysis. Based on the PRNS's error detection capability, the error correction method is introduced by adding one redundant moduli to the error detection module, following which the implementation of such GF(2^{163}) multiplier with error correction capability is presented.

Shown from the hardware implementation results of the error detection multiplier designs, different combinations of number of channels and the channel length of the PRNS architectures yield different synthesis result. For the same dynamic range, smaller number of channels provides smaller overhead in hardware with the cost of the increased channel length. In addition, from the error coverage analysis, the increased channel length helps to improve the multiple error-detecting rate. The implementation of the fault tolerant design has shown significant overhead in hardware, which is mainly because of the reduplicative SRC conversion circuits. The proposed GF(2^{163}) error detection multiplier and error correction multiplier is suitable for the ECC designs that require high level of security where hardware consuming is not a main issue. In the following two chapters, the proposed PRNS architecture will be applied to the AES application, where the field length is small and the hardware overhead is manageable.

Chapter 5
Low Area Design of the AES

5.1 Introduction

The Rijndael cipher algorithm, introduced by Vincent Rijmen and Joan Daemen, was selected as the Advanced Encryption Standard (AES) by the National Institute of Standards and Technology (NIST) in 2000. In the following year, this algorithm became the Federal Information Processing Standard FIPS-197 [6]. As the AES has been widely adopted for various applications from high-end computers to low power portable devices, numerous hardware architectures to implement the AES were proposed to meet different requirements. Typical examples are high-throughput design and low-area design. The former aims to achieve highest operating frequency and throughput. The latter devotes most efforts to minimize the size of the design and lower the power consumption.

FPGA platforms have emerged recently as a viable low cost alternative to ASICs in many domains which have seen a trend of using FPGAs for actual production rather than just prototyping due to their advantages in terms of reconfigurability (flexibility with low cost); shorter time to market (easy to debug and short development cycle); increasingly efficient fabric (advanced processes); and also the fact that FPGA manufacturers provide mask programmed versions of their technologies. Optimal FPGA designs for cryptography are particularly desirable when scalability or compatibility with different applications is required in secure applications or when design IP protection is sought. Area (and energy) optimality is the most challenging in

the design space. Therefore low resource, but with acceptable performance, cryptography primitives such as the AES on FPGA are key enablers for many applications to implement strong security or protection.

In this chapter, a compact AES FPGA encryption core is proposed based on an iterative round-looping architecture as in [75] where the shifting operations are re-designed to exploit the FPGA fabric in Spartan 3 and Spartan 6 generations to reduce overall area and improve speed. The proposed design only occupies 184 slices of a XC3S50 FPGA, achieves a throughput of 36.5Mbps; on a Spartan 6 XC6SLX4 FPGA, this design occupies 80 slices with a throughput of 58.13Mbps. Since most useful modes (OFB[1], CTR[2] and CFB[3]) [76, 77, 78] can all provide data encryption and decryption using only an encryption-primitive, it was decided to implement a design that performs AES encryption only, as this is the minimum requirement for three useful modes. To the authors' knowledge, the proposed design is believed to be the smallest memory free FPGA implementation of the AES encryption in literature.

For the mathematical background information of the AES algorithm, the reader is referred to Chapter 2.5. The rest of this chapter is organized as follows: firstly, previous work on the AES is reviewed as references, then the detailed design of the proposed AES architectures is presented with the FPGA specific optimizations, the hardware results and comparisons with previous reported works are given before conclusions are drawn.

[1] OFB, Output Feedback Mode
[2] CTR, Counter Mode
[3] CFB, Cipher Feedback Mode

5.2 Review of the Previous AES Designs

Speed and resource consumption are the key system requirements to implement the AES algorithm, which drove most of the previous works focus either on high throughput or low area.

Pipelined (or sub-pipelined) and loop-unrolled architectures with large data path (usually 128-bit) are usually adopted to enable high-speed in the throughput focused AES designs. Typical examples can be found in [79, 80], where their designs achieve the throughput over 20Gbps. The drawback of high throughput designs is that they occupy large hardware resources and consume high power; in addition, these architectures are not suitable for feedback modes in some operations [80, 81].

Round-looping and sub-function-sharing are the mostly used technique to implement the low area AES. The data path is also reduced from 128-bit to 32-bit or even 8-bit to decrease the parallelism of operations therefore reduces the hardware consumption. Typical 32-bit low area AES design can be found in [82, 83, 84], where the smallest one uses 222 slices and 9600-bit block RAM (totally equivalent to 522 slices) and achieves 166Mbps throughput. Some 8-bit designs have better performance in term of area, such as the ASIP (application specific instruction processor) design proposed in [80], it only uses 124 slices and 4480-bit block RAM (totally equivalent to 264 slices), achieves a throughput of 2.2Mbps. To the author's knowledge, the work in [80] is so far the smallest FPGA implementation of the AES in the literature. The work presented in this chapter only consumes 184 slices and does not require any block memory; furthermore the proposed design achieves a much higher (36.5Mbps) throughput compared with the smallest design in [80].

5.3 The Proposed Design of the Low Area AES

5.3.1 FPGA Specific Optimizations

This design explores the FPGA fabric technology in Spartan 3 and Spartan 6 generations, which can configure the LUT in one slice as a shift register instead of using the available flip-flops of each slice, to optimise the design by improving the performance of the shifting operations in the AES.

In such LUT based shift registers, the shift-input operations are synchronous with the clock, and output length can be selected dynamically using variable taps [85, 86]. The example schematic diagram of such addressable shift register (SRL16 in Spartan 3 FPGAs) is shown in Figure 5-1. The 32-bit LUT based shift registers, SRL32, in the Spartan 6 FPGA family, using the same architecture, doubles the shift register length with an additional address signal.

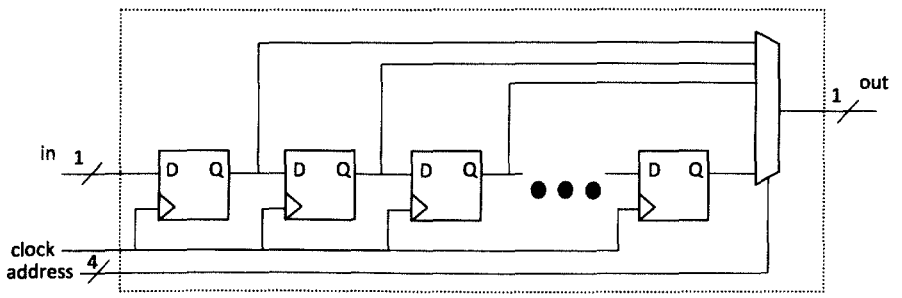

Figure 5-1: LUT Based Addressable 16-bit Shift Register (SRL16)

Using SRL16 to implement an 8-bit wide, 16-bit long shift register only requires 4 slices in a Spartan 3 device, which leads to great cost saving. In addition, the address taps give a convenient way to select the wanted output and are very suitable for the ShiftRow implementation. Similar technology exists in Spartan 6 devices, LUTs can be configured as either 16-bit shift registers (SRL16) or 32-bit shift registers (SRL32).

5.3.2 Top Level Architecture

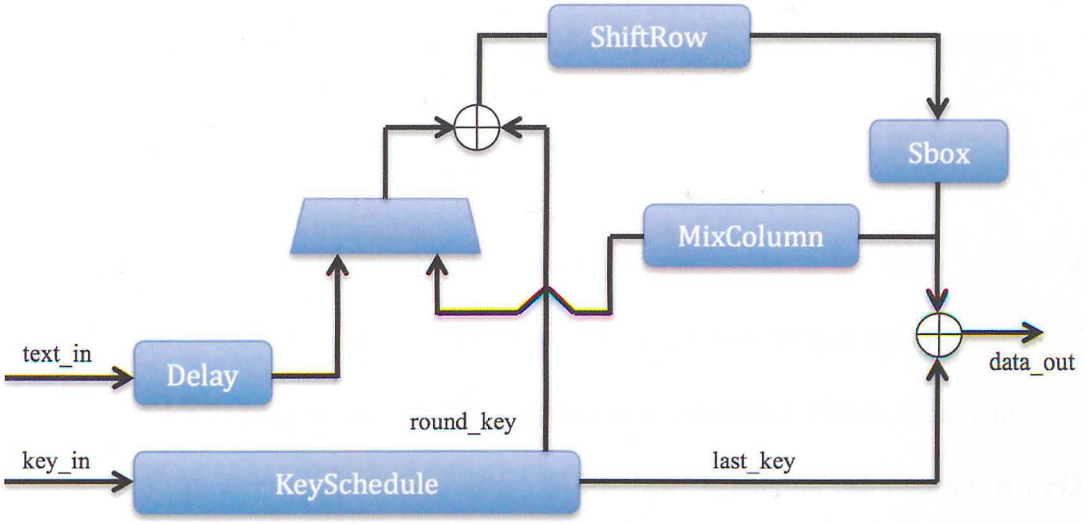

Figure 5-2: AES Encryption Core Architecture for 8-bit Data Path

The main architecture of this design adopts an iterative pipelined-round-looping architecture with an 8-bit data path. All the sub-functions are performed in parallel in order to reduce the number of clock cycles. The design supports 128-bit keys and requires 160 clock cycles to finish encrypting one 128-bit block. The top-level architecture is shown in Figure 5-2.

It mainly consists of five sub-function blocks: ShiftRow, Sbox, MixColumn, KeySchedule and the input Delay. In this work, the ShiftRow, Delay and KeySchedule blocks are redesigned and constructed using SRL16/32 to minimize the consumption of the number of FPGA slices. Both input for the plain text (*text_in*) and the key (*key_in*) are required to be 8-bit, the final *data_out* is given as an 8-bit vector as well. The KeySchedule block performs the KeyExpansion transformation and generates new roundkeys every 16-cycles. The input Delay block uses a shift register to propagate the plain text with 4 cycles delay, in order to synchronize with the other operands (the RoundKey) of the AddRoundKey operation, which is implemented

using the XOR operation on the left side of Figure 5-2. The final XOR operation (on the right side of Figure 5-2) performs the AddRoundKey transformation for the last round as the last round transformation bypasses the MixColumn transformation.

5.3.3 Design of ShiftRow

The ShiftRow operation rearranges the location of each byte in the block text. In the proposed architecture, two sets of SRL16 are cascaded to construct an addressable 32-bit shift register to perform the shifting operation in Spartan 3 FPGA, where only one single SRL32 is used as an addressable 32-bit shift register in the Spartan 6 FPGA design; eight such shift registers works in parallel with sharing address taps. Detailed structure is shown in Figure 5-3. In Spartan 6 FPGA, eight SRL32 are working in parallel to perform the shifting operation Figure 5-4.

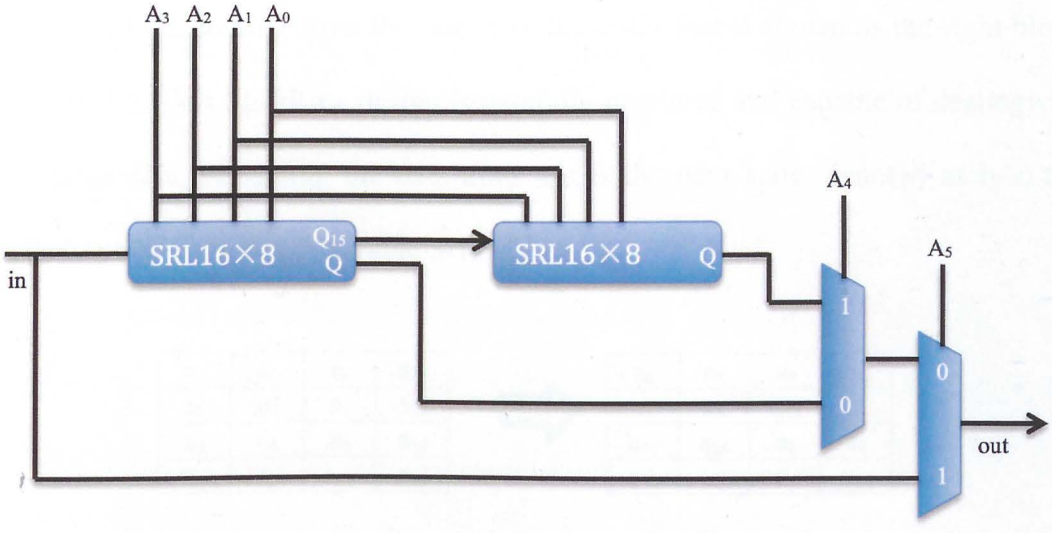

Figure 5-3: SRL16 Based ShiftRow

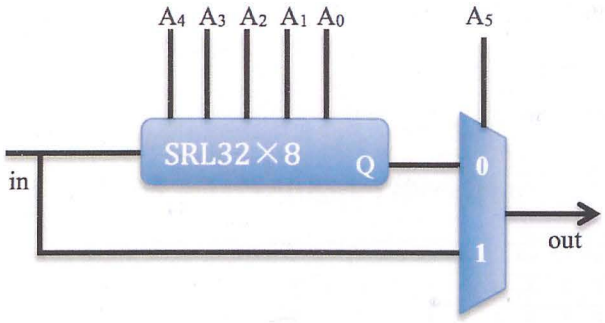

Figure 5-4: SRL32 Based ShiftRow

While data is shifted through the shift registers, the address taps select the reordered date to the output. The ShiftRow block is naturally pipelined; it has 12 clock cycles latency, but when fully filled with data it can deal with continuous data input. Here is an example to demonstrate how ShiftRow module works. To perform the shifting operation shown in Figure 5-5, 12 clock cycles are required to shift the first three columns of data (a_0 to a_{11}) into the shift register. After 12 cycles, every cycle, there will be one byte coming from the output in the order that is shown as the right block in Figure 5-5. This ShiftRow design is naturally pipelined and capable of dealingwith continuous data. Assuming the data after a_i's is the next state, denoted as b_i's, the detailed operation is listed in Table 5-1.

a_0	a_4	a_8	a_{12}
a_1	a_5	a_9	a_{13}
a_2	a_6	a_0	a_{14}
a_3	a_7	a_{11}	a_{15}

⇨

a_0	a_4	a_8	a_{12}
a_5	a_9	a_{13}	a_1
a_{10}	a_{14}	a_2	a_6
a_{15}	a_3	a_7	a_{11}

Figure 5-5: Demonstration of the ShiftRow Transformation

As it can be seen from Table 5-1, the address taps A_1 and A_0 are constantly '1' during the shifting operation, hence a 4-bit state machine is constructed to generate the correct address signal for A_5 to A_2 and it repeats the states every 16 clock cycles. This SRL16/32 based ShiftRow block, including the address generating state machine,

only occupies 20 slices and can be operated at 265MHz solely on a Spartan III XC3S50 FPGA.

Table 5-1: ShiftRow Operation

t	in	R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}	R_{12}	R_{13}	R_{14}	R_{15}	R_{16}	R_{17}	R_{18}	R_{19}	R_{20}	R_{21}	R_{22}	R_{23}	A_5	A_4	A_3	A_2	A_1	A_0	out
1	a_{12}	a_{11}	a_{10}	a_9	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	X	X	X	X	X	X	X	X	X	X	X	X	0	0	1	0	1	1	a_0
2	a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	X	X	X	X	X	X	X	X	X	X	X	0	0	0	1	1	1	a_5
3	a_{14}	a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	X	X	X	X	X	X	X	X	X	X	0	0	0	0	1	1	a_{10}
4	a_{15}	a_{14}	a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	X	X	X	X	X	X	X	X	X	1	X	X	X	X	X	a_{15}
5	b_0	a_{15}	a_{14}	a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	X	X	X	X	X	X	X	X	0	0	1	0	1	1	a_4
6	b_1	b_0	a_{15}	a_{14}	a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	X	X	X	X	X	X	X	0	0	0	1	1	1	a_9
7	b_2	b_1	b_0	a_{15}	a_{14}	a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	X	X	X	X	X	X	0	0	0	0	1	1	a_{14}
8	b_3	b_2	b_1	b_0	a_{15}	a_{14}	a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	X	X	X	X	X	0	0	1	1	1	1	a_3
9	b_4	b_3	b_2	b_1	b_0	a_{15}	a_{14}	a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	X	X	X	X	0	0	1	0	1	1	a_8
10	b_5	b_4	b_3	b_2	b_1	b_0	a_{15}	a_{14}	a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	X	X	X	0	0	0	1	1	1	a_{13}
11	b_6	b_5	b_4	b_3	b_2	b_1	b_0	a_{15}	a_{14}	a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	X	X	0	1	0	0	1	1	a_2
12	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	a_{15}	a_{14}	a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	X	0	0	1	1	1	1	a_7
13	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	a_{15}	a_{14}	a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	0	0	1	0	1	1	a_{12}
14	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	a_{15}	a_{14}	a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	0	1	0	1	1	1	a_1
15	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	a_{15}	a_{14}	a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8	a_7	a_6	a_5	a_4	a_3	a_2	0	1	0	0	1	1	a_6
16	b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	a_{15}	a_{14}	a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8	a_7	a_6	a_5	a_4	a_3	0	0	1	1	1	1	a_{11}
17	b_{12}	b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	a_{15}	a_{14}	a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8	a_7	a_6	a_5	a_4	Repeating						b_0

5.3.4 Design of Sbox

The Sbox performs the SubBytes transformation. The SubBytes transformation is the only non-linear operation among all AES transformations, which contains a multiplicative inversion calculation followed by an affine transformation. The crux of implementing the SubBytes is the implementation of the multiplicative inversion over $GF(2^8)$. There are mainly three ways to implement the SubBytes transformation that have been appeared in literature:

- Direct calculation: it computes the multiplicative inversion over $GF(2^8)$ directly using either multiplication and square algorithm or Itoh and Tsujii's algorithm. This method often appears in the design of the general $GF(2^m)$ processor for cryptography, a typical example is given in [12].

- LUT approach: it uses a table looking up method to substitute a byte with its

substitution value from an 8-bit × 256 pre-stored table. This table is given in the official AES paper FIPS-197 [6], as shown in Table 5-2:

Table 5-2: SubBytes Look Up Table

		y														
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

- Composite field arithmetic approach: to simplify the inversion calculation, this approach first decomposes the GF(2^8) into field GF(2^4) (or decomposes even further using GF(2^2)), then computes the multiplicative inversion over the smaller field, in the end maps the inversion back to GF(2^8) before the affine transformation. This method is first proposed in [14] and has been further developed in many works for both low cost AES designs [80, 83, 88] and the sub-pipelined high throughput AES designs [79].

In the proposed low area AES design, the pure combinational logic constructed composite field arithmetic approach is adopted to achieve the lowest possible slice consumption of the FPGA implementation of the SubBytes transformation. The mathematic background of the composite field arithmetic is referenced in [14, 79, 87, 88] for interested reader. The implementation of the forward SubBytes transformation using composite field arithmetic is demonstrated in Figure 5-6:

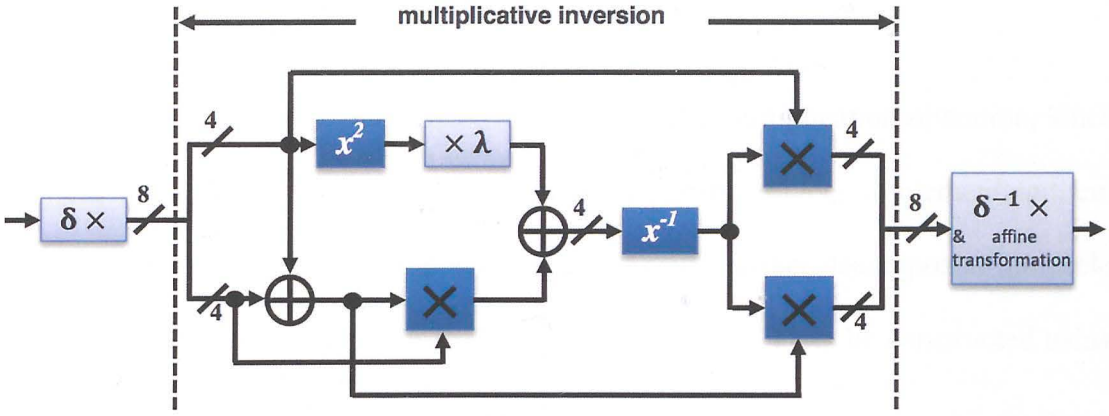

Figure 5-6: SubBytes Transformation using Composite Field Arithmetic [79]

The multiplicative inversion section of the above figure shows the GF(2^8) inversion computation in the decomposed field GF((2^4)^2). In order to perform an equivalent inversion in composite field arithmetic, additional isomorphic mapping function (denoted as $\delta\times$) and its inverse (denoted as $\delta^{-1}\times$) need to be applied to map the representation of an element in GF(2^8) to its composite field and vice versa. Both δ and δ^{-1} are represented using 8×8 binary matrix (as following) that are generated from the irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$ over GF(2^8):

$$\delta = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}, \delta^{-1} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

In Figure 5-6, the x^2, × λ, $x^{-1}$and × represent square operation, multiplication with constant λ, multiplicative inversion, multiplication operation over GF(2^4) respectively.

The multiplying by constant λ operation can be implemented as following: assuming the input and output are denoted in binary vector format as $(a_3a_2a_1a_0)$ and $(c_3c_2c_1c_0)$ respectively, then

$$c_3 = a_0 \oplus a_2, \qquad c_2 = a_0 \oplus a_1 \oplus a_2 \oplus a_3,$$
$$c_1 = a_3, \qquad c_0 = a_2$$

For the square operation, multiplicative inversion and multiplication operation, since they are all over GF(2^4), there are two options to implement them: implementing them directly in GF(2^4) or using composite field arithmetic further decomposing the field into GF(2^2). According to [79, 87], the squarer over GF(2^4) can be constructed using very simple XORs, so it is implemented as:

$$c_3 = a_3, \qquad c_2 = a_2 \oplus a_3,$$
$$c_1 = a_1 \oplus a_2, \quad c_0 = a_0 \oplus a_1 \oplus a_3$$

The multiplication over GF(2^4) can be performed directly using the logic that is given in [87] as: assuming $(a_3 a_2 a_1 a_0)$ and $(b_3 b_2 b_1 b_0)$ are the input, $(c_3 c_2 c_1 c_0)$ is the output.

$$c_0 = a_0 b_0 \oplus a_3 b_1 \oplus (a_2 \oplus a_3) b_2 \oplus (a_1 \oplus a_2) b_3$$
$$c_1 = a_1 b_0 \oplus (a_0 \oplus a_3) b_1 \oplus a_2 b_2 \oplus a_1 b_3$$
$$c_2 = a_2 b_0 \oplus a_1 b_1 \oplus (a_0 \oplus a_3) b_2 \oplus (a_2 \oplus a_3) b_3$$
$$c_3 = a_3 b_0 \oplus a_2 b_1 \oplus a_1 b_2 \oplus (a_0 \oplus a_3) b_3$$

The composite field arithmetic based multiplication over GF(2^4) is given in [79], the implementation is demonstrated in Figure 5-7:

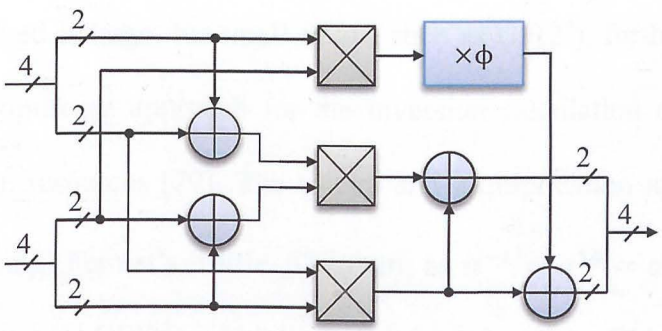

Figure 5-7: GF(2^4) Multiplication Using Composite Field [79]

In Figure 5-7, the $\times\phi$ block is constructed according to the following equation:

$$c_0 = a_1, \qquad c_1 = a_1 \oplus a_0$$

The GF(2^2) multiplication block (the ⊠ block) is constructed according to the following equations:

$$c_0 = (a_1 \oplus a_0)(b_1 \oplus b_0) \oplus a_0 b_0$$
$$c_1 = a_1 b_1 \oplus a_0 b_0$$

From the experimental result, the synthesis results show both approaches of realizing the GF(2^4) multiplication occupy the same FPGA area, which is 7 slices over a Spartan III device, however, when the entire SubBytes block is synthesised, the result shows the composite field approach has better performance in terms of area, which is 35 slices to 41 slices that uses the direct GF(2^4) multiplication over the same FPGA platform.

To implement the multiplicative inversion in Figure 5-6, [79] shows three approaches, which are composite field approach, square and multiplication approach and the truth table approach. The composite field approach, firstly convert GF(2^4) to the composite field GF((2^2)^2) and then calculate the inversion over GF(2^2) followed by a conversion back to GF(2^4) to obtain the inversion over GF(2^4). However, though the composite field decomposition can reduce the hardware complexity significantly when the order of the field involved is large, for small fields, such as GF(2^4), further decomposition may not be the optimum approach for the inversion calculation as the conversion circuit costs more resources [79]. The square and multiplication approach calculate the inversion using Fermat's Little Theorem as $\alpha^{-1} = \alpha^{14} = \alpha^2 \cdot \alpha^{2^2} \cdot \alpha^{2^3}$ over GF(2^4), it requires two GF(2^4) multipliers and three squarers. This approach shows largest area consumption and combinational delay among the three approaches. The truth table approach uses the equations, that are derived from the truth table of calculating the GF(2^4) inversion, to directly compute the multiplicative inversion. As

it has been shown in [79], this approach achieves smallest number of gates with smallest critical path. Thus, the calculation of the multiplicative inversion over GF(2^4) is performed as following equations in the proposed design:

$$c_3 = a_3 \oplus a_3a_2a_1 \oplus a_3a_0 \oplus a_2$$
$$c_2 = a_3a_2a_1 \oplus a_3a_2a_0 \oplus a_3a_0 \oplus a_2 \oplus a_2a_1$$
$$c_1 = a_3 \oplus a_3a_2a_1 \oplus a_3a_1a_0 \oplus a_2 \oplus a_2a_0 \oplus a_1$$
$$c_0 = a_3a_2a_1 \oplus a_3a_2a_0 \oplus a_3a_1 \oplus a_3a_1a_0 \oplus a_3a_0 \oplus a_2 \oplus a_2a_1 \oplus a_2a_1a_0 \oplus a_1 \oplus a_0$$

To complete the byte substitution of the SubBytes using composite field arithmetic, a isomorphism conversion from GF((2^4)^2) back to GF(2^8) followed a affine transformation that is defined by the AES algorithm is required. The isomorphism and affine transforms may be combined into one single transform [88]. This results in the following matrix:

$$\delta^{-1}A(x) = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

The proposed design of the SubBytes operation occupies 35 slices over the Spartan III device, where the LUT based implementation occupies 8 × 256 = 2k bits block memory or 64 slices for distributed memory.

5.3.5 Design of MixColumn

The MixColumn design adopts the architecture that is introduced in [75]. The architecture is demonstrated in Figure 5-8:

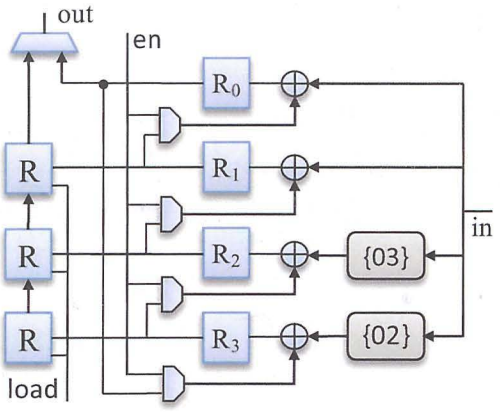

Figure 5-8: MixColumn Using 8-bit Data Path

In this module, one column of a state date is treated at a time in four clock cycles. Each clock cycle a new byte is fed to the unit, the four registers (R_0 to R_3) store the intermediate results of the MixColumn calculation. Every four cycles, upon the completion, the 32-bit output is fed to the parallel-to-serial converter (parallel load shift register R's), after which the output of the MixColumn block becomes 8-bit per cycle. The {03} and {02} block performs the multiplication by constant 03_{HEX} and 02_{HEX} over GF(2^8) respectively. They are constructed according to the following equations:

$$\{02\} \cdot a(x) = a_6x^7 + a_5x^6 + a_4x^5 + (a_3 + a_7)x^4 + (a_2 + a_7)x^3 + a_1x^2 + (a_0 + a_7)x + a_7$$

$$\{03\} \cdot a(x) = (a_6 + a_7)x^7 + (a_5 + a_6)x^6 + (a_4 + a_5)x^5 + (a_3 + a_4 + a_7)x^4 + (a_2 + a_3 + a_7)x^3 + (a_1 + a_2)x^2 + (a_0 + a_1 + a_7)x + (a_0 + a_7)$$

This MixColumn architecture is naturally pipelined and capable of dealing with continuous data streaming with a latency of 4 clock cycles. The detailed operational procedure of the MixColumn transformation that uses the above architecture is demonstrated in Table 5-3:

Table 5-3: 8-bit MixColumn Operations

	T=0	T=1	T=2	T=3
R_0	c_0	$c_0 \oplus c_1$	$\{03\}c_0 \oplus c_1 \oplus c_2$	$\{02\}c_0 \oplus \{03\}c_1 \oplus c_2 \oplus c_3$
R_1	c_0	$\{03\}c_0 \oplus c_1$	$\{02\}c_0 \oplus \{03\}c_1 \oplus c_2$	$c_0 \oplus \{02\}c_1 \oplus \{03\}c_2 \oplus c_3$
R_2	$\{03\}c_0$	$\{02\}c_0 \oplus \{03\}c_1$	$c_0 \oplus \{02\}c_1 \{03\}c_2$	$c_0 \oplus c_1 \oplus \{02\}c_2 \oplus \{03\}c_3$
R_3	$\{02\}c_0$	$c_0 \oplus \{02\}c_1$	$c_0 \oplus c_1 \oplus \{02\}c_2$	$\{03\}c_0 \oplus c_1 \oplus c_2 \oplus \{02\}c_3$

Where T indicates the clock cycles, R_i is the intermediate result, c_i is the number of bytes in a column of a state.

5.3.6 Design of KeySchedule

The KeySchedule expands the original cipher key to derive the roundkeys for each AddRoundKey transformation. There are two approaches of implementing it: pre-computing approach and on-the-fly key generating approach. In the first approach, all roundkeys are pre-generated and stored before the AES encryption process starts; typical example can be found in literature [80, 81, 82, 83, 89]. The drawback of this approach is the consumption of a considerable amount of storage space, however, this approach is suitable for high throughput focused designs and is more energy efficient in the long run, if the key is not changed [75]. The on-the-fly approach generates the roundkeys alongside of the round transformations; it requires area only for a single KeyExpansion mechanism without extra storage. The on-the-fly approach is usually adopted by low area designs, examples can be found in [84, 90, 91]. It should be noted that when using the same cipher key to encrypt more than one block of plain text, this approach continuously repeats the work already done, which results in more power consumption.

The proposed low area AES design adopts an on-the-fly architecture with 8-bit data path. The architecture is shown in Figure 5-9:

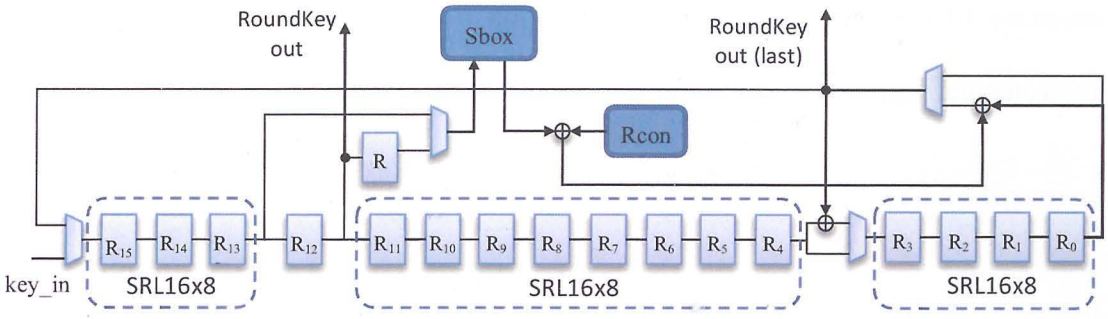

Figure 5-9: On-the-fly KeySchedule with 8-bit Data Path

As it can be seen from Figure 5-9, an additional Sbox block is introduced to the KeySchedule, that is because the Sbox block in the round transformation is fully occupied while encrypting data due to the pipelined architecture, which makes it un-shareable with other operations. The introduction of the extra Sbox does not significantly increase the total area due to its compact design. It is designed using the same method that has been introduced in Chapter 5.3.4.

The Rcon block generates the round constant and can be constructed using an 8-bit linear feedback shift register (see Figure 5-10). The shift register is initialized as 01_{HEX} and shifts one time for each roundkey generation.

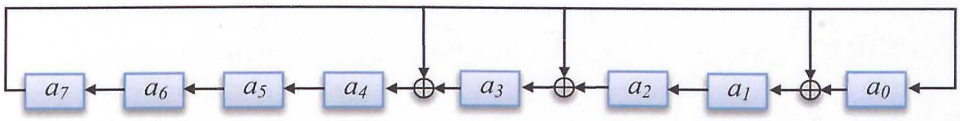

Figure 5-10: Rcon Generation

In the KeySchedule unit, three sets of SRL16 based shift registers are used as shown in Figure 5-9 to optimize the performance on FPGA platform, where the SRL16's address taps are fixed to a constant value. It takes 16 clock cycles for the KeySchedule module to generate a new roundkey and store them in the intermediate registers. The *RoundKey_out* forward roundkeys out to the AddRoundKey transformation of the normal round operations; the *Last_RoundKey_out* is only used

for the last AddRoundKey operation. This KeySchedule design occupies 81 slices on a Spartan III XC3S50 device with a highest possible operating frequency of 46.035MHz.

5.3.7 Design of Top Level Control

Due to the fully pipelined looping architecture, this design does not require complex control signals. A simple 8-bit binary counter, which counts from 0 to 160, is build to generate the enable signals to the built-in state machines that are located in the ShiftRow, MixColumn blocks. Followed by a simple decoder, this counter also generates the data flow selection signals in the KeySchedule block.

5.4 Implementation Results and Comparisons

The synthesis, placing and routing of the complete design were done using Xilinx ISE 11.1. Table 5-4 shows the synthesis results of the proposed FPGA AES encryption core. This design is the smallest memory free FPGA implementation of the AES encryption core to date. Comparisons with other low cost work are also listed in Table 5-4.

Table 5-4: Low Area AES Design Synthesis Results Comparisons

	Chodowiec & Gaj [82]	Rouvroy et al [83]	Pramstaller et al [84]	T.Good & M.Benaissa [80]	Picoblaze based [80]	Yong Sung Jeon et al [92]	**This design**
FPGA	Spartan II XC2S30-6	Spartan III XC3S50-4	Virtex-E XCV1000E	Spartan II XC2S15-6	Spartan II XC2S15-6	Spartan II XC2S30-6	**Spartan III XC3S50-5**
Clock Frequency (MHz)	60	71	161	67	90	66	**45.642**
Data path	32	32	32	8	8	8	**8**
No. of Clock Cycles	44	46	92	3691	13546	352	**160**
Slices	222	163	1125	124	119	258	**184**
No. of Block RAMs	3	3	0	2	2	0	**0**
Block RAM Size (kbits)	4	18	0	4	4	0	**0**
Bits of block RAM used	9600	34176	0	4480	10666	0	**0**
Total Equivalent Slices	522	1231	1125	264	452	258	**184**
Throughput (Mbps)	166	208	215	2.2	0.71	24	**36.5**
Throughput/slice (kbps/slice)	318	169	191	8.3	1.9	93	**198**
Summary	Best speed/area	-	Fastest	ASIP	Software	-	**Smallest**

It can be seen from Table 5-4, that this design also achieves much higher throughput than the listed 8-bit ASIP and PicoBlaze designs but is not as high as the 32-bit designs due mainly to the narrowed data path and more clock cycles.

Table 5-5 shows the hardware and performance comparison with the Helion Company's Tiny AES core family, which are announced to be the smallest commercial AES solutions [93]. To have a fair comparison, the proposed architecture has been implemented on to a Spartan 6 FPGA using 32-bit LUT based shift registers (SRL32). This design only occupies 80 slices of a Spartan 6 device with the throughput doubled to the commercial AES core.

Table 5-5: Synthesis Results Comparisons with Industry Products

	FPGA	MAX Throughput	Slices	Block RAM
Tiny AES cores [93]	Spartan 3E	30 Mbps	166	1
	Spartan 6	29 Mbps	91	0
This work	**Spartan 3**	**36.5** Mbps	**184**	**0**
	Spartan 6	**58.13** Mbps	**80**	**0**

5.5 Function Testing

This design has been tested and verified using the ModelSim based testbench. The test vectors are provided by the official AES paper FIPS-197 [6], where the detailed step-by-step test vectors can also be found.

5.6 Conclusions

In this chapter, a compact AES encryption core on FPGA is presented. Thanks to the specific features brought by Spartan 3/6 FPGA platform, an AES design with the lowest area is achieved. The low-cost implementation and moderate throughput make this solution practically suitable for security focused low resource applications. Although this design is for AES encryption only, it still can satisfy most applications, for it is estimated that 25% additional area consumption will be required to add the decryption functionality to this design.

The next chapter will look into the application of the PRNS architecture onto the AES, especially to explore the error control capability brought by the PRNS and its applications in the AES designs.

Chapter 6
Error Detecting AES using PRNS

6.1 Introduction

A new method using PRNS is introduced in this chapter to protect the AES against faults attacks. By using PRNS, the byte based AES operations over GF(2^8) are decomposed into several parallel operations that use its residues over smaller fields. Three GF(2^4) irreducible polynomials are selected as the moduli set for the chosen PRNS, including a redundant modulus to achieve error detection. Three GF(2^4) AES cores are constructed individually according to the chosen moduli.

This PRNS architecture brings several advanced features to AES design from the scope of anti-side-channel analysis. The proposed error detecting scheme can detect 100% single bit errors and up to 4 bit errors that occur in a single GF(2^4) AES core, and 93.75% multiple errors across different AES cores for each byte based operation. The error detection mechanism is constructed using a simple XOR-AND network, which is quite low in hardware cost. In addition, the original AES operations are distributed across three GF(2^4) AES cores, each of which has its own data path, so it adds to the AES design built-in resistance against probing attacks. Furthermore, a unique SBox look-up-table (LUT) is constructed for each GF(2^4) AES core where redundant information is added; hence it boosts the confusion level of the system. Detailed design information is shown in Chapter 6.3. Two different architectures that apply PRNS to the AES are demonstrated in this chapter, one is based on a 32-bit data path AES, the other uses an 8-bit data path round-looping architecture to implement the AES.

Hardware overhead is compared and analysed for the different architectures before the conclusion is drawn. Error coverage analysis and comparisons with other AES error detection schemes are given in Chapter 6.5.

6.2 Review of Existing AES Error Detection Scheme

As the AES has been widely adopted for different applications, higher reliability of the AES design is required. In recent years, numerous attacks have been introduced to break cryptographic systems and extract secret information via side-channel-analysis by analysing or manipulating the observations of physical characteristics of the electronic cryptographic system. Typical examples are timing attacks [94], power attacks [95], electromagnetic radiation attacks [96] and fault attacks [97, 98].

Prior work has shown that even a single transient error occurring during the AES round operations will very likely result in a large number of errors in the final data [68]. In addition, a few attack scenarios have shown that the AES is quite vulnerable to fault attacks [68, 69, 70, 71, 72]. Hence it is necessary to provide error detection mechanisms to the AES design to achieve higher level of reliability and security.

There are several approaches to achieve error detection for cryptographic systems. Generic solutions are duplication and repeated computation, however these solutions either double hardware overhead or latency and they are not protective against permanent faults. Error detecting codes are widely used by engineers to implement error proof designs. In [99], an overview of error detecting codes based protection mechanisms for AES implementations can be found. There are mainly two solutions: parity code based schemes [68, 100, 101] and residue code based schemes [102, 103]. The parity-based methods have low hardware overhead but are weak for multiple faults detection; the residue code based error detection schemes have good multiple faults coverage but are weak in single fault detection and become very complicated and hardware consuming when predicting the residue codes for non-linear operations such as the SubBytes operation in the AES.

6.3 PRNS based Error Detection AES

6.3.1 Top Architecture and PRNS Representation

To implement the PRNS architecture, three GF(2^4) AES cores are individually constructed. They perform the AES transformations (both round transformations and key generation) using the original data's residue representation. According to PRNS theory, an arbitrary GF(2^8) element can be uniquely represented using its two GF(2^4) residues. A redundant GF(2^4) AES core is introduced to construct the illegitimate range for error detection (see Chapter 4.2). The error detection mechanism converts the residue representation back to normal representation and performs the overflow detection.

Apart from the non-linear SubBytes transformation, the only operation that has the potential of exceeding the dynamic range that is covered by the given PRNS, among all the AES transformations is the MixColumn transformation, where it contains multiplying by constant 02_{HEX} and 03_{HEX} operation over GF(2^8) for the AES encryption. A clever approach is introduced in Chapter 6.3.3 to achieve the overflow prediction for the MixColumn transformation using partial conversion method that has been introduced in Chapter 3.3. To avoid the non-linear transformation and the conversion to-and-from PRNS representation, the SubBytes transformation over PRNS is implemented using LUT approach, the detailed information will be given in Chapter 6.3.2. The top architecture of the proposed design is shown in Figure 6-1.

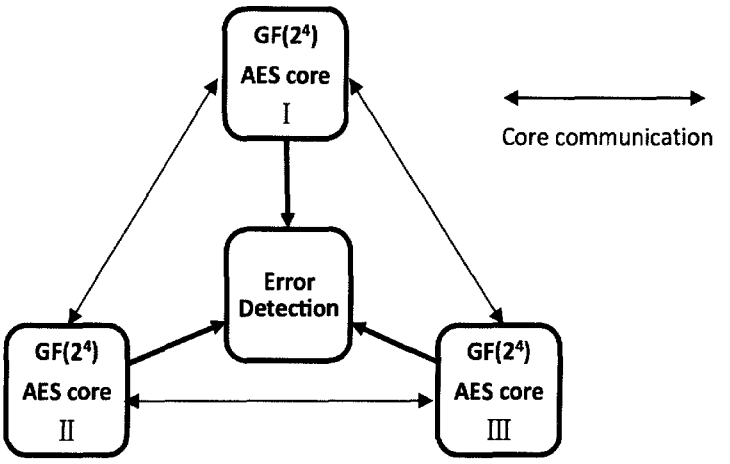

Figure 6-1: Top Architecture of the PRNS based AES

The PRNS representation of the original data is used for each AES core. Irreducible polynomials $m_1: x^4 + x + 1, m_2: x^4 + x^3 + 1$ and $m_3: x^4 + x^3 + x^2 + x + 1$ are selected to compute the residues for each core. So each byte in the original block is represented using three residues. After the modular operation, the original 128-bit block becomes three 64-bit blocks. Each 64-bit block is processed by a GF(2^4) AES core. An example is demonstrated in Figure 6-2.

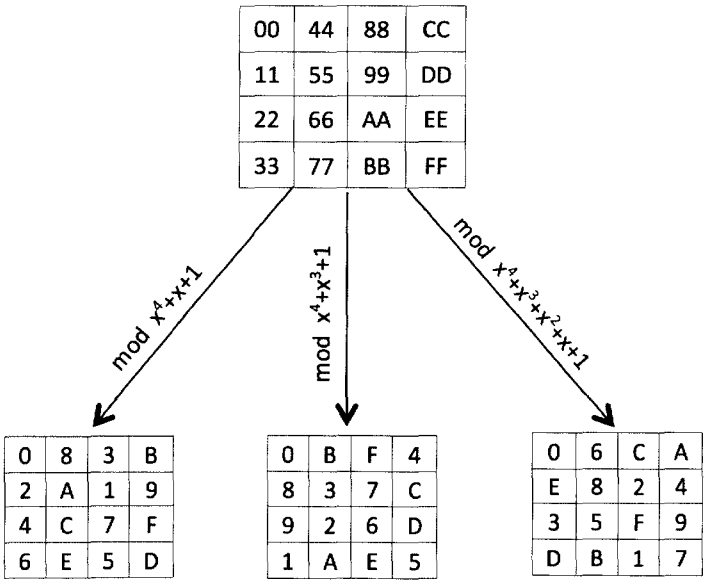

Figure 6-2: State Block in PRNS Representation

6.3.2 SubBytes Transformation using PRNS

SubBytes transformation is the only non-linear operation in the AES, which computes the multiplicative inverse of each byte of the state block followed by an affine transformation. It is very expensive in hardware if it is computed directly in PRNS form; therefore a judicious LUT approach is adopted to implement SubBytes (Sbox) for the proposed design. Each AES core contains a unique Sbox LUT. It is constructed by firstly generating the original Sbox's residue representation according to the selected irreducible polynomial, then re-arranging each entry's location in the table according to the new address to form a new table. Each table has 256 entries, each of which contains a 4-bit word. In the selected redundant PRNS, each 8-bit word is represented using three 4-bit residues. Due to the existence of the redundant residue, any two residues from the three-residue representation can uniquely represent an 8-bit word. Each Sbox LUT uses two residues as its address.

Here is an example to demonstrate how this table-look-up method works:

$$SubBytes([AA]) = [AC]$$

This result is looked up from the original offical Sbox in Table 5-2, where [AA] and [AC] is normal hex decimal numbers. The PRNS representation of $[AA]$ is $(7, 6, F)$. Now we use its residues as address for table look up:

$$TABLE\ I(6,7) = 1, TABLE\ II(F, 6) = 0, TABLE\ III(7, F) = 9$$

So, in the selected PRNS,

$$SubBytes(7, 6, F) = (1, 0, 9)$$

If the result $(1, 0, 9)$ is converted back to its normal representation using SRC algorithm, the result is equal to $[AC]$, which is the same as the result from the original Sbox, indicating the validity of the proposed PRNS Sbox LUT.

Table 6-1: Sbox over GF(2^4) $x^4 + x + 1$ for Core I

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	9	9	2	F	D	B	B	5	B	F	0	1	4	3	F	9
1	D	5	9	0	9	3	4	F	4	F	6	0	4	8	9	1
2	A	2	E	5	A	4	6	C	2	4	D	6	6	D	F	8
3	F	0	0	2	0	1	B	1	2	F	E	1	C	E	1	0
4	8	3	6	5	0	5	C	D	D	6	4	7	5	0	9	F
5	4	2	2	7	7	1	2	5	4	0	3	E	C	5	6	4
6	A	B	7	E	D	E	5	1	7	0	C	C	7	7	E	C
7	E	F	4	E	D	3	8	2	1	9	8	A	2	B	5	B
8	2	7	9	3	A	0	F	4	5	5	8	D	5	0	8	B
9	8	D	F	D	B	A	1	9	6	1	5	C	B	A	3	3
A	4	8	7	6	7	3	2	F	D	4	D	2	4	7	7	7
B	3	E	0	C	8	5	9	E	8	3	3	D	1	C	8	D
C	8	B	9	F	E	0	6	9	E	6	3	C	C	1	B	9
D	5	7	B	B	C	2	6	8	D	E	C	8	7	3	A	E
E	2	A	A	A	2	B	1	0	A	A	4	6	A	8	6	F
F	1	6	3	3	6	B	C	7	1	A	E	C	9	9	A	F

Table 6-2: Sbox over GF(2^4) $x^4 + x^3 + 1$ for Core II

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	7	9	E	9	A	D	8	1	7	6	7	3	7	7	1	E
1	5	1	9	7	D	9	6	F	6	5	5	2	3	8	9	E
2	6	B	A	9	8	C	8	D	D	7	C	A	5	8	8	9
3	E	F	3	6	1	2	9	D	E	D	F	1	0	A	4	C
4	8	A	3	2	8	9	5	5	7	C	E	2	9	0	D	3
5	B	6	1	E	3	F	C	2	C	1	8	5	3	1	E	4
6	5	2	2	8	E	A	B	B	F	1	7	2	4	B	0	A
7	C	1	F	2	F	F	A	D	7	5	A	B	B	D	8	4
8	C	B	9	6	5	7	C	5	2	C	B	0	E	D	A	2
9	F	D	B	6	6	4	9	D	3	1	B	D	0	3	2	8
A	B	D	2	4	4	4	1	4	E	0	3	A	F	3	3	6
B	4	6	7	0	0	8	9	8	E	D	F	0	A	0	4	6
C	8	4	9	1	3	2	9	0	A	F	0	4	4	7	A	C
D	3	B	5	6	A	7	5	C	E	0	1	1	F	6	B	5
E	B	A	4	2	C	E	E	6	5	E	7	1	D	3	7	F
F	4	C	0	F	0	3	**0**	C	6	2	9	5	F	C	8	B

Table 6-3: Sbox over GF(2^4) $x^4 + x^3 + x^2 + x + 1$ for Core III

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	0	0	9	4	B	C	0	F	E	2	D	D	1	A	8	3
1	E	0	9	5	6	8	A	6	3	A	7	D	3	F	2	4
2	E	2	B	3	E	8	6	0	8	1	5	7	E	2	6	E
3	B	8	3	7	1	3	3	6	5	4	1	3	2	0	E	C
4	C	7	0	8	A	F	7	7	8	C	F	F	A	4	0	1
5	F	D	7	A	4	8	3	4	0	5	6	5	D	4	4	B
6	F	D	F	B	D	B	C	4	F	0	4	F	5	5	1	D
7	5	A	B	B	2	9	0	3	7	9	5	2	2	1	2	**9**
8	6	F	C	6	A	2	4	B	E	E	6	3	C	6	1	F
9	3	B	C	F	7	7	C	D	F	1	5	B	C	C	A	0
A	3	9	1	9	7	C	D	2	4	D	4	7	E	D	9	B
B	8	2	8	5	B	5	5	8	5	1	9	A	1	3	2	C
C	6	2	9	8	C	D	1	D	9	6	8	2	6	0	A	3
D	B	E	6	7	1	7	7	9	9	E	9	8	8	E	B	0
E	D	C	1	A	A	5	A	1	6	A	0	D	8	B	E	4
F	2	5	3	7	4	9	F	4	0	9	6	E	C	E	F	A

It can be noticed that, in this approach, all the table look up operations are done by using PRNS representations as addresses, and the results form the SubBytes transformation are in PRNS as well. Thus, no conversion circuit is needed; this can also lower the potential of information leaking from the conversion circuit.

Three LUTs are operated individually in parallel. Attackers need to trace the information from at least two tables to obtain enough information, thus this adds more difficulties to crack the system.

Furthermore, as it can be seen from the contents repetitively in the tables, even if an attacker obtains the LUT output, it is not easy for the attacker to trace the input address, because there are several different entries with the same value. Hence, the PRNS Sbox has higher level of confusion compared with the original Sbox LUT. It enhances the security level of the proposed architecture.

6.3.3 MixColumn Transformation using PRNS

MixColumn transformation can be seen as a matrix multiplication operation. It contains a few multiplying by constant x and $x + 1$ operations. It is easy to implement these operations in normal weighted polynomial representation, however, as the dynamic range that is covered by the selected PRNS is only 8-bit for this design (not including the redundant modulus, which is used for error detection), overflow may occur and a modular reduction over GF(2^8) using the field generating polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$ is needed to correct the result.

Here introduces a clever way of implementing the multiplication by x and $x + 1$ operations to avoid complicated conversion between PRNS and normal representation and restrict the operations to the dynamic range that is covered by the chosen PRNS. This approach adopts the partial conversion method that has been described in Chapter 3.3.

The main idea is derived from the observation that the highest possible degree of a GF(2^8) element that is multiplied by x and $x + 1$ will be 8, which indicates the possible overflow will be 1 bit only. The overflow can be predicted directly from the operand before the multiplication is done. Only if the highest degree of the operand is equal to 7, after the multiplication, the highest degree will exceed 7 and cause overflow. The prediction mechanism partially converts the MSB, which is the highest degree bit, from the PRNS representation to determine if a modular reduction over GF(2^8) is needed.

Taking multiplying by x as an example:

For a GF(2^8) element, denoted in its binary vector format as $A(a_7a_6a_5a_4a_3a_2a_1a_0)$

and its PRNS representation $\langle A_1\rangle_{m_1}, \langle A_2\rangle_{m_2}, \langle A_3\rangle_{m_3}$, the multiplication is performed as follows:

Where $m_1: x^4 + x + 1$, $m_2: x^4 + x^3 + 1$, $m_3: x^4 + x^3 + x^2 + x + 1$ and

$$m: x^8 + x^4 + x^3 + x + 1$$

$$(a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0) \cdot x \bmod m \quad (1)$$

$$= [a_7x^8 + (a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0) \cdot x] \bmod m$$

$$= (a_7x^8 \bmod m) + (a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0) \cdot x$$

$$= \boxed{a_7(x^4 + x^3 + x + 1)}_{①}$$

$$+ \boxed{(a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0) \cdot x}_{②}$$

From the above equation, none of ① and ② will exceed the defined dynamic range, and the addition over binary field will not cause over flow, so it can be computed correctly using the selected PRNS.

In the PRNS, taking the field GF(2^4) defined by $m_1: x^4 + x + 1$ as an example (in this field $x^7 = x^4 + x + 1$, which will be used in the following equation transformations), after the conversion to the PRNS using m_1, equation (1) yields:

$$①: [a_7(x^4 + x^3 + x + 1) \bmod m_1] = a_7x^3 \quad (2)$$

$$②: (a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0) \cdot x \bmod m_1$$

$$= (a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 + a_7x^7) \cdot x \bmod m_1$$

$$= \{[(A + a_7x^7) \cdot x] \bmod m_1\}$$

$$= [A_1 + a_7(x^4 + x + 1)] \cdot x \bmod m_1 \quad (3)$$

Using equation (2) and (3), the multiplying by x operation can be done using PRNS without causing any overflow. It is only required to convert one bit to normal representation. Using the partial conversion method simplifies the calculation of a_7, because there is no carry effect over binary fields.

Multiplying by $x + 1$ operation can apply the same method, because it generates the same overflow as multiplying by x operation.

6.3.4 Other Transformations using PRNS

Both ShiftRow and AddRoundKey are linear operations and will not cause any overflow problems, so can be implemented using PRNS directly. The round constant Rcon for the proposed PRNS architecture is generated from the normal Rcon generation over GF(2^8) followed by a to PRNS conversion.

6.3.5 Error Detecting Mechanism

Error detection performs the SRC algorithm using partial conversion method followed by overflow detection. For one byte, the overflow only occurs in the most significant 4 bits, so only partial conversion is needed. Partial conversion brings several advantages to the design. It lowers the potential of leaking information, simplifies the conversion circuit and saves hardware resources. The detailed partial conversion method is demonstrated in Chapter 3.3. Overflow detection checks if there are '1's in the most significant 4 bits, a simple 4-bit AND gate can be used for this.

6.4 Design of GF(2^4) AES Core

6.4.1 32-bit Data Path AES using PRNS

The first attempt of constructing an AES encryption core using PRNS architecture adopts a 32-bit data path and column transformation based approach to trade-off hardware consumption and throughput. Due to the use of a PRNS representation, each GF(2^4) AES core uses a 16-bit data path. The architecture introduced in [104] is adopted.

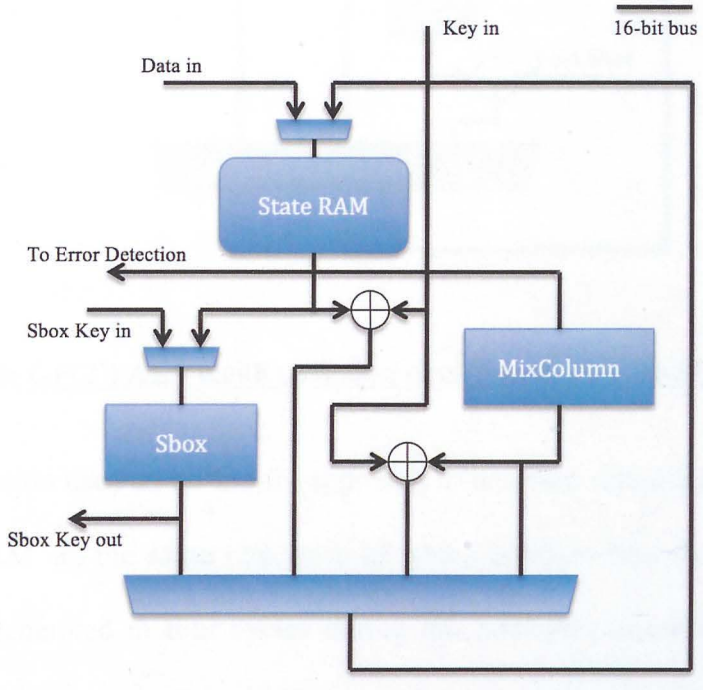

Figure 6-3: GF(2^4) AES Encryption Core Architecture for 32-bit Data Path

The encryption core mainly consists a StateRAM, Sbox, Mixcolumn and several XORs for AddRoundKey transformation. The StateRAM is constructed using four 8x4bit dual-port RAM, where address is individually generated in order to perform the ShiftRow transformation. The Sbox contains four PRNS Sbox LUTs, which can perform SubBytes transformation for a column. Sbox is also shared with the

KeyExpansion. To process a block plaintext for a round, it needs four cycles to finish SubBytes and ShiftRow operations and another four cycles to perform MixColumn and AddRoundKey. The design of the 32-bit MixColumn applies the substructure sharing method that has been introduced in [79].

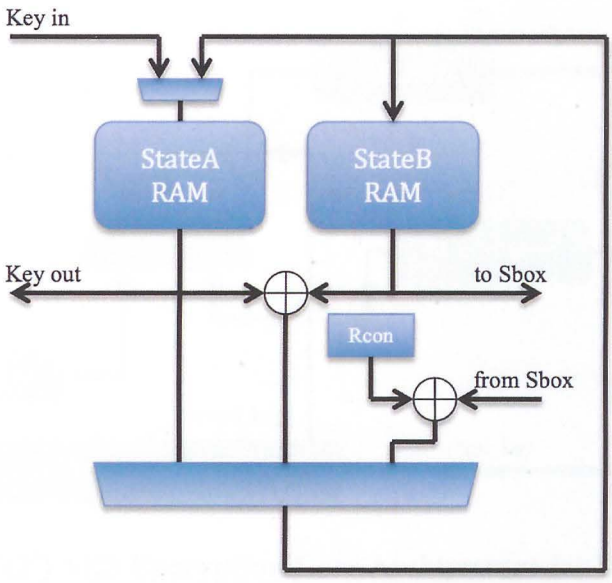

Figure 6-4: GF(2^4) AES KeyExpansion Architecture for 32-bit Data Path

The KeyExpansion uses an on-the-fly approach to generate round keys. StateA RAM and StateB RAM are the same size, each of which contains four 4x4bit RAM. New round key is generated in four cycles during the SubBytes operation and stored in StateA RAM. During the AddRoundKey transformation, the round key is sent to the encryption core column by column; at the same time, the round key is transferred to StateB RAM preparing to generate the key for the next round. At the beginning of each round operation, an extra cycle is needed by KeyExpansion to perform SubBytes transformation for roundkeys.

In the proposed 32-bit PRNS AES, four conversion circuits work in parallel to detect the errors for four bytes (32-bit) in one column.

6.4.2 8-bit Data Path AES using PRNS

The second attempt to implement the error detection AES using PRNS adopts a low area iterative round-looping architecture with an 8-bit data path that has been introduced in Chapter 5.

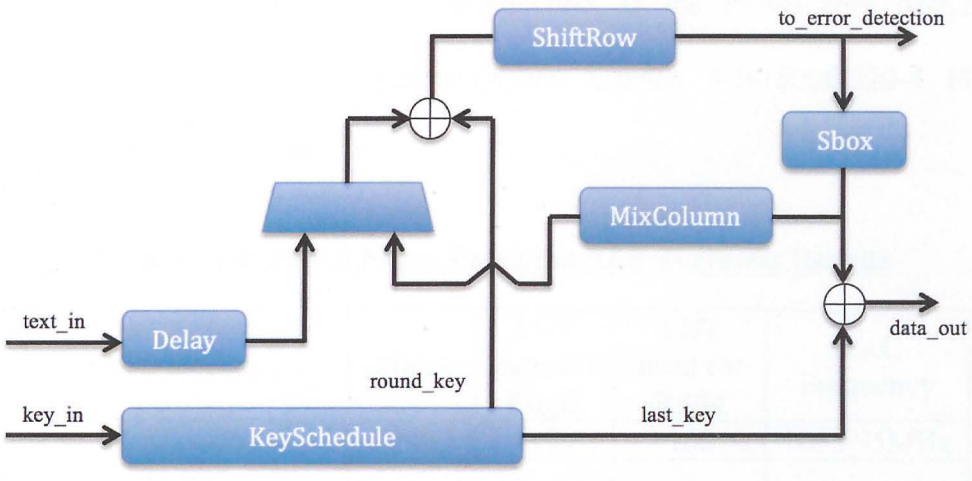

Figure 6-5: GF(2^4) AES Encryption Core Architecture for 8-bit Data Path

It mainly consists of five sub-function blocks: ShiftRow, Sbox, MixColumn, KeySchedule and the input Delay (Figure 6-5). The data path for this core is 4-bit. The error detection is performed after the ShiftRow transformation.

In this architecture, apart from the Sbox, which is constructed using the proposed PRNS table look up method, the rest blocks are using the same architecture that has been introduced in Chapter 5.

6.5 Hardware Implementation and Results

Table 6-4 shows the synthesis results of the proposed PRNS error detection AES. To the authors' knowledge, this is the first attempt for the AES design using such PRNS error detection scheme. To enable a fair comparison, a normal 32-bit AES and a normal 8-bit AES, which adopts the same architecture as the PRNS core design, are implemented onto the same platform (Xilinx Spartan 3-3s1500fg320-4 FPGA). Comparisons are listed below:

Table 6-4: PRNS Error Detection AES Synthesis Results

AES Design	Slices	LUT used for Logic	LUT used for RAM	Max. Frequency
32-bit AES	590	1190	160	103.595MHz
32-bit Non-redundant PRNS AES	711	1345	160	101.204MHz
32-bit Redundant PRNS AES	1068	2095	240	103.767MHz
8-bit AES	243	466	0	62.344MHz
8-bit Redundant PRNS AES	385	739	0	51.794MHz

It can be seen from the above table, as expected, due to the use of the same architecture for the PRNS cores, the PRNS designs achieve similar operating frequency to the normal AES designs. The maximum operating frequency in the 32-bit PRNS AES that is slightly higher than the normal 32-bit AES design is due to different routing delays of the FPGA implementation. The operating frequency of the 8-bit PRNS AES design is lower than the normal AES is mainly because the PRNS conversion and error detection circuits add extra logic to the critical path, therefore increase the maximum combinational delay, and lower the frequency. In terms of added hardware overhead, the 32-bit PRNS AES design adds 81% overhead and the 8-bit PRNS AES adds about

58% overhead compared with the standard AES, which is quite acceptable for a multiple error detection scheme. The reason why the proposed 32-bit PRNS AES has larger overhead percentage than the 8-bit PRNS AES is that in the 32-bit design four conversion circuits are build to deal with 32-bit data, whereas only one conversion is needed in the 8-bit design.

6.6 Error Coverage Analysis and Comparison

For the AES byte operation, the proposed error-detecting scheme is capable of detecting 100% single bit errors and 100% single core errors (where error occurs only in one core, up to 4-bit multiple errors). If multiple faults occur across different cores, the probability of detecting the error by this scheme is (only those errors that do not cause overflow will be missed)

$$\frac{2^{12} - 2^{8}}{2^{12}} = 93.75\%$$

Comparisons with other error detection schemes are shown in the following Table 6-5. As it can be seen, though having quite large hardware overhead, the advantages brought by this design are apparent. Firstly it not only covers 100% single bit faults, but has excellent multiple faults coverage as well; secondly, unlike those code-predicting schemes [100, 101, 103], PRNS error detection can be performed directly without extra predicting mechanisms, so it adds no extra clock cycles overhead.

Table 6-5: AES Error Detection Scheme Comparison

Method	Single Fault Detection	Multiple Fault Detection	Hardware Overhead	Delay Overhead
Single parity bit [100, 101]	100%	no	+7.4%	+6.4%
Multiple parity bits *(n=16)*[68]	100%	Double faults masked with $P \propto \frac{1}{n}$	+20%	-
Linear+non-linear codes [102]	Weak	Good	+35%	-
Non-linear r-bit codes (r=28) [103]	Good, missed with $P \propto 2^{-2r}$	Good, missed with $P \propto 2^{-2r}$	+77%	+15%
Redundant PRNS (32-bit)	100%	93.75%	+81%	-
Redundant PRNS (8-bit)	100%	93.75%	+58%	-

6.7 Conclusions

In this Chapter, the PRNS implementations of the AES have been advocated for error detection and protection against side-channel and fault attacks. The proposed error-detecting scheme yields very good error coverage; Furthermore, the distribution and parallelism characteristic of a PRNS architecture itself yields intrinsic resistance to some side-channel attacks. A proposed PRNS based Sbox implementation is believed to offer higher level of confusion.

The PRNS architecture brings a new design methodology to implement the AES. Besides the error detection capability, the non-error-detection PRNS AES can provide improved side-channel-attack resistance with only 20% hardware overhead, which is quite remarkable from the security and hardware implementation point of view. In addition, due to the flexible selection of PRNS generating polynomials and number of PRNS channels, random PRNS channel selection can be used to bring more randomization and confusion to the system, which can be a strong weapon against power analysis.

According to the attempted implementations, an 8-bit architecture offers the most optimal option in terms of added overhead.

Chapter 7
Conclusions and Further Work

7.1 Conclusions

This thesis has concentrated on research dealing with the Polynomial Residue Number System (PRNS) over the domain of $GF(2^m)$, for applications within the sub domain of cryptography. The first part of the thesis focused mainly on the arithmetic side of the PRNS implementations; in Chapter 3, the designs of the PRNS based $GF(2^m)$ multiplier were presented. A number of different architectures spanning different moduli types have been proposed together with the corresponding hardware implementation results. It was found that the conversion circuit (mainly the from PRNS conversion circuit) was the source of the main overhead in such PRNS multiplier. To overcome this obstacle, a novel conversion and modular reduction method has been introduced to the PRNS architecture, namely partial modular reduction method. A new implementation of such multiplier adopting the partial modular reduction method has been presented together with its hardware results. From the comparison of different PRNS multiplier architectures, the partial modular reduction method enables great reduction in the use of area and the combinational delay, thereby improving the performance, which makes such PRNS multiplier feasible, as shown, for extensive cryptography primitives such as ECC using curve K-163. Chapter 4 introduced the error detection and error correction capability that is provided by the PRNS architecture. The mathematic proof of the error detection capability was given in the first place, followed by a detailed example and the

implementation of an 8-bit GF multiplier. The implementation details of such error detection multiplier over GF(2^{163}) also has been presented together with the error coverage analysis. Based on the PRNS's error detection capability, the proposed error correction (fault tolerance) method has been introduced by adding one extra redundant moduli to the error detection module. A detailed description of a fault tolerant GF(2^{163}) multiplier, together with its FPGA synthesis results, have been given in Chapter 4 as a demonstration of the proposed error correction method. It was shown from the hardware implementation results of the error detection multiplier designs, that different combinations of number of channels and the channel length of the PRNS architectures yield different synthesis result. For the same dynamic range, smaller number of channels provided smaller overhead in hardware with the cost of the increased channel length. In addition, from the error coverage analysis, it was concluded that the increased channel length helps to improve the multiple error-detecting rate. The implementation of the fault tolerant design has shown significant overhead in hardware, which is mainly due to the reduplicative SRC conversion circuits. A proposed GF(2^{163}) error detection multiplier and error correction multiplier is shown to be suitable for ECC designs that require high level of security where hardware consumption is not a main issue.

The second part of the thesis concentrated on the application side of the PRNS implementations. The Advanced Encryption Standard (AES) algorithm has been selected as the target application. Before applying the PRNS architecture to the AES design, Chapter 5 presented a very low area AES design on an FPGA platform as a reference to the PRNS based AES. By exploiting the specific features brought by Spartan 3/6 FPGA fabric, which are the LUT based shift registers, the proposed AES core has achieved the lowest area ever reported on an FPGA platform without using

any block memory. The design only requires 184 slices on a Xilinx Spartan 3 (XC3S50) device, and 80 slices on a Spartan 6 (XC6SLX4) device while achieving throughputs of 36.5Mbps and 58.13Mbps respectively. The low-cost implementation and moderate throughput make this solution practically suitable for security focused low resource applications. In Chapter 6, the PRNS implementations of the AES have been advocated for error detection and protection against side-channel and fault attacks. The proposed PRNS error-detecting scheme, which applies the PRNS architecture to the AES core, yields very good error coverage; furthermore, the distribution and parallelism characteristic of a PRNS architecture itself yields intrinsic resistance to some side-channel attacks. A proposed PRNS based Sbox implementation is believed to offer higher level of confusion. The PRNS architecture brings a new design methodology to implement the AES. Besides the error detection capability, the non-error-detection PRNS AES can provide improved side-channel-attack resistance with only 20% hardware overhead, which is quite remarkable from the security and hardware implementation point of view. The proposed low area error detection AES, which is based on the low area AES design that has been presented in Chapter 5, only occupies 385 slices on a Spartan III device with the maximum operating frequency of 51.794MHz, which is particularly suitable for area constrained cryptography designs, embedded systems and SoC (System on Chip) designs, when higher level of security are required.

7.2 Further Work

The PRNS architecture over GF(2^m) brings an entire new design methodology for GF(2^m) circuits and applications. There are a number of directions that can be explored further.

- Fault Tolerant Cryptography Applications:

  By adding an extra redundant channel to the proposed design in Chapter 6, the PRNS based AES is capable of providing error correcting capability with an estimated overhead of $1/3$. Furthermore, the proposed PRNS GF(2^{163}) fault tolerant multiplier (Chapter 4) is suitable for ECC applications using curve K-163 over binary field to fight against fault attacks.

- Randomization and Masking for Cryptography Applications:

  Due to the flexible selection of PRNS generating polynomials and number of PRNS channels, random PRNS channel selection can be used to bring more randomization and confusion to the system, which can be a strong weapon against power analysis. It is possible to add redundant PRNS channels to the target application and use the redundant channels to process trash information. Due to the similarity of the PRNS channel constructions and the added 'noise' by the redundant channels, this PRNS architecture with will provide natural masking capability to the internal cryptography transformations, by which the security of the crypto-system is enhanced. It has to be noted that, the difficulty locates in the generation of the conversion circuit for the randomly selected channel generating polynomials. Possible solution may be found in the scope of software and hardware combined architecture, where the software side pre-computes the constant values that

are needed according to the randomly selected irreducible polynomials, the hardware side provides a generic architecture, which does not vary to different channel selections.

- Scalable Designs:

  Different combination of the number of channels and the channel length of the PRNS architecture will provide different dynamic range coverage. This property can be further explored for scalable designs where the same hardware architecture is capable of dealing with altered field length. For example, there are five curves over binary field that are recommended by the NIST (National Institute of Standards and Technology), for field length equal 163, 233, 283, 409, and 571. It is possible to find a PRNS set to cover the dynamic range that is provided by the largest field, and use partial of the PRNS set to process the smaller field.

- System on Chip Designs:

  The proposed low area AES design (in Chapter 5) and the low area error detecting AES design (in Chapter 6) are suitable for low cost SoC designs that aim to implement one or several particular crypto-protocols on a single chip. Works can be done to find the utilities for the proposed AES cores in such designs.

Chapter 8 References

[1] M. C. Yang, J. L. Wu. "A New Interpretation of Polynomial Residue Number System," *IEEE Transactions on Signal Processing*. Vol.42, No.8, August 1994.

[2] J. Chu, "Public Key Cryptography using Residue Number Systems on FPGA," MSc dissertation, *The University of Sheffield*, Sheffield, UK, 2007.

[3] M. Ciet, M. Neve, E. Peeters, J. Quisquater, "Parallel FPGA implementation of RSA with Residue Number Systems: can side-channel threats be avoided?" *IEEE Midwest Symposium on Circuits and Systems*, Dec. 2003, Egypt.

[4] M. G. Parker and M. "Benaissa, Fault-Tolerant Linear Convolution using Residue Number Systems," *Proc of ISCAS'94*, London, Vol 2, pp 441-445, May 1994.

[5] S. B. Wicker, *Error Control Systems for Digital Communication and Storage*, Prentice-Hall International, 1995, ISBN: 0132008092.

[6] National Institute of Standards and Technology (NIST), *Advanced Encryption Standard (AES) Federal Information Processing Standards Publication 197* (FIPS PUB 197), Nov. 2001.

[7] National Institute of Standards and Technology (NIST), *Digital Signature Standard (DSS) Federal Information Processing Standards Publication 186* (FIPS PUB 186-3), June. 2009.

[8] R. E. Blahut, *Fast Algorithms for Digital Signal Processing*, Addison-Wesley, Reading, Massachusetts, 1985

[9] C. C. Wang, D. Pei, "A VLSI design for computing exponentiation in GF(2^m)

and its application to generate pseudorandom number sequences." *IEEE Transactions on Computers*, C-39(2):258-262. Feb. 1985.

[10] T. A. Gulliver, M. Serra, V. K. Bhargava, "The generation of primitive polynomials in GF(q) with independent roots and their application for power residue codes, VLSI teasing and finite field multipliers using normal bases." *Int. J. electronics*, 71(4):559-576, 1991.

[11] L. Rudolf, N. Harald, *Finite Fields (2^{nd} ed)*, Cambridge University Press, 1997, ISBN 0-521-39231-4.

[12] W. M. Lim, "Design of Application Specific Instruction Set Processors for the Domain of GF(2^m)," PhD thesis, *The University of Sheffield*, Sheffield, UK, 2004.

[13] E. Mastrovito, "VLSI Architectures for Computations in Galois Fields," Ph.D thesis, *Linkoeping University*, Linkoeping, Sweden, 1991.

[14] C. Paar, "Efficient VLSI architecture for bit-parallel computations in Galois field," Ph.D. dissertation, *Institute for Experimental Mathematics, University of Essen*, Essen, Germany, 1994.

[15] P. A. Scott, S . E. Tavares, and L. E. Peppard, "A fast VLSI multiplier for GF(2^m)," *IEEE J. Select. Areas Commun.*, vol. SAC-4, pp. 62-66, Jan. 1986.

[16] C. Y. Lee, C. W. Chiou, J. M. Lin and C. C. Chang, "Scalable and systolic Montgomery multiplier over GF(2^m) generated by trinomials," *IET Circuits Devices Syst.*, 2007, 1, (6), pp. 477–484

[17] Jain, S.K. Parhi, K.K., "Low latency standard basis GF(2^m) multiplier and squarer architectures," *ICASSP-95*, May 1995, Detroit, MI, USA.

[18] C. Paar, "A New Architecture for a Parallel Finite Field Multiplier with Low Complexity Based on Composite Fields," *IEEE Transactions on Computers*, Vol. 45, No. 7, July 1996.

[19] H. Wu, M. A. Hasan, "Low Complexity Bit-Parallel Multipliers for a Class of Finite Fields," *IEEE Transactions on Computers*, Vol. 47, No.8, pp. 883-887, 1998.

[20] H. Wu, "Bit-Parallel Finite Field Multiplier and Squarer Using Polynomial Basis," *IEEE Transactions on Computers*, Vol. 51, No.7, July 2002.

[21] G. Orlando, C. Paar, "A super-serial Galois fields multiplier for FPGAs and its application to public-key algorithms," *FCCM 1999*, Napa Valley, CA, USA. April 1999

[22] L. Song, K. K. Parhi, "Low-Energy Digit-Serial/Parallel Finite Field Multipliers," *Journal of VLSI Signal Processing*, Vol. 19, pp. 149–166, 1998

[23] G. Orlando, C. Paat, "Squaring architecture for GF(2^m) and its applications in cryptographic systems," *Electronics Letters*, Vol. 36, No. 13, June 2000 .

[24] H. Eberle, A. Wander, N. Gura, S. Chang-Shantz, "Architectural extensions for elliptic curve cryptography over GF(2^m)," July 2004, Available at: https://research.sun.com/sunlabsday/docs.2004/Micro.pdf

[25] H. Brunner, A. Curiger, and M. Hofstetter, "On computing multiplicative inverses in GF(2^m)," *IEEE Transactions on Computers*, Vol. 42, No. 8, pp. 1010–1015, 1993.

[26] J. Wang and A. Jiang, "An Area-Efficient Design for Modular Inversion in GF(2m)," *IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, pp. 1496–1499, 2006.

[27] C. K. Koc, T. Acar, "Montgomery multiplication in GF(2^k)," *Designs, Codes and Cryptography*, 14(1):57–69, April 1998.

[28] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, 44(170): 519–521, April 1985.

[29] H. Wu, "Montgomery Multiplier and Squarer in GF (2^m)," *Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems*, 2000, ISBN: 3-540-41455-X.

[30] F. J. Taylor, "Residue arithmetic: a tutorial with examples," *Computer*, Vol 17, No. 5, pp.50-63, May 1984.

[31] L. L. Yong, A. T. Se, *Fleeting Footsteps: tracing the conception of arithmetic and algebra in ancient China (Revised edition)*, Singapore, River Edge NJ: World Scientific Publishing Co., 2004.

[32] M. A. Soderstrand, et al., *Residue Number System arithmetic: modern applications in digital signal processing*, New York: IEEE Press, 1986.

[33] A. Svoboda, M. Valach, "Operational Circuits," *Stroje na Zpracovani Informaci*, Sbornik III, Nakl. CSAV, Prague, 1955, pp.247-295.

[34] D. M. Schinianakis, A. P. Kakarountas, T. Stouraitis, "A new approach to elliptic curve cryptography: an RNS architecture," *IEEE MELECON 2006*, May 16-19, Benalmádena (Málaga), Spain.

[35] A. P. Riyaz, "A study and implementation of parallel-prefix modular adder architechtures for the Residue Number System," Ph.D. dissertation, *The University of Sheffield*, Sheffield, UK, 2006.

[36] J. C. Bajard, L. Imbert, P. Y. Liardet, "Leak resistant arithmetic," *LIRMM*, Tech. Rpt. No. 03021. Oct. 2003.

[37] N. S. Szabo, R. I. Tanaka, *Residue arithmetic and its applications to computer technology*, New York: McGraw Hill, 1967.

[38] B. Parhami, *Computer arithmetic: algorithms and hardware designs*, Oxford: Oxford University Press, 2000.

[39] W. K. Jenkins, *Handbook for digital signal processing, ed.* S. K. Mitra and J. F. Kaiser. NY: John Wiley & Sons, Inc., 1993

[40] T. V. Vu, "Efficient implementation of the Chinese remainder theorem for sign detection and residue decoding," *IEEE Transactions on Computers*, Vol. C-34, No.7, pp. 646-51, Jul. 1985.

[41] S. J. Piestrak, "Design of high-speed residue-to-binary number system converter based on Chinese Remainder Theorem," *in Proc. IEEE Int. Conf. Comput. Design (ICCD 1994)*, Cambridge, MA USA, pp. 508-511, Oct. 1994.

[42] A. D'Amora, et al, "Reducing power dissipation in complex digital filters by using the quadratic residue number system," *in Conf. Record IEEE 34th Asil. Conf. Signals, Syst. Comput. (ACSSC 2000)*, Vol. 2, Pacific Grove, CA USA, pp. 879-83, Oct-Nov. 2000.

[43] W. L. Freking, K. K. Parhi, "Low-power FIR digital filters using residue arithmetic," *Conf. Record IEEE 31th Asil. Conf. Signals, Syst. Comput. (ACSSC 1997)*, Vol. 1, Pacific Grove, CA USA, pp. 739-43, Nov. 1997.

[44] T. Stouraitis, V. Paliouras, "Considering the alternatives in low-power design," *IEEE Circuits Devices Mag.*, Vol. 17, No. 4, pp. 22-9, Jul. 2001.

[45] A. Skavantzos, F. J. Taylor, "On the Polynomial Residue Number System," *IEEE Transctions on Signal Processing*, Vol. 39, No. 2, February 1991.

[46] M. G. Parker, M. Benaissa, "GF(p^m) multiplication using polynomial residue number systems," *IEEE Transactions on Circuit and Systems II: Analog and Digital Signal Processing*, Vol.42, No. 11, pp. 718-721, 1995.

[47] A. Skavantzos, T. Stouraitis, "Polynomial Residue Complex Signal Processing," *IEEE Transactions on Circuits and Systems - II*, Vol 40, No 5, pp. 342 - 344, May 1993.

[48] A. Halbutoğullari, C. K. KOC, "Parallel multiplication in GF(2^k) using polynomial residue arithmetic," *Designs, Codes and Cryptography*, Vol. 20 No. 2, pp. 155–173, June 2000.

[49] P. J. Ashenden, *The Designer's Guide to VHDL*, Morgan Kaufmann, 1996, ISBN: 1558602704

[50] D. L. Perry, *VHDL, 3rd Edition ed, Computer Science Series*. New Delhi: McGraw-Hill International Editions, 1999.

[50] Mentor-Graphic, "ModelSim", Available at: http://www.model.com/

[51] Irreducible and primitive polynomials over GF(2), Available at: http://fchabaud.free.fr/English/default.php?COUNT=2&FILE0=Poly&FILE1=GF(2)

[52] Y. Han, P. C. Leong, P. C. Tan, J. Zhang, "Fast algorithms for elliptic curve cryptosystems over binary finite field," In *Advances in Cryptology — ASIACRYPT '99*, LNCS 1716, pp. 75–85. Springer Verlag, 1999.

[53] M. A. Hasan, "Look-up table-based large finite field multiplication in memory constrained cryptosystems," *IEEE Transactions on Computers*, 49(7):749–758, July 2000.

[54] J. C. Lopez Hernandez, R. Dahab, "High-speed software multiplication in IF (2^m)," *In Progress in Cryptology — INDOCRYPT 2000*, LNCS 1977, pp. 203–212, Springer Verlag, 2000.

[55] N. P. Smart, "A comparison of different finite fields for elliptic curve cryptosystems," *Computers and Mathematics with Applications*, 42(1-2):91–100, July 2001.

[56] S. V. Bharathwaj, K. L. Narasimhan, "An alternate approach to modular multiplication for finite fields GF(2^m) using Itoh Tsujii algorithm," *IEEE-NEWCAS Conference*, pp.103–105, June 2005.

[57] M. Hütter, J. Großschädl and G. A. Kamendje, "A Versatile and Scable Digit-Serial/Parallel Multiplier Architecture for Finite Field GF(2^m)," *The International Conference on Information Technology: Computers and Communications*, 2003.

[58] N. S. Chang, C. H. Kim, Y. H. Park, and J. Lim, "A Non-Redundant and Efficient Architecture for Karatsuba-Ofman Algorithm," *In Information Security, 8th International Conference, ISC 2005*, Singapore, September 20-23, 2005, Proceedings, Vol. 3650 of Lecture Notes in Computer Science, pages 288–299. Springer, 2005.

[59] S. Erdem, C. K. Koc, "A Less Recursive Variant of Karatsuba-Ofman Algorithm for Multiplying Operands of Size a Power of Two," *In 16th IEEE Symposium on Computer Arithmetic (Arith-16 2003)*, Santiago de Compostela, Spain, pp. 28–35, 15-18 June 2003.

[60] F. Rodrıguez-Henrıquez, C. K. Koc, "Parallel Multipliers Based on Special Irreducible Pentanomials," *IEEE Transactions on Computers*, 52(12):1535–1542, 2003.

[61] V. Serrano-Hernandez, F. Rodrıguez-Henrıquez, "An FPGA Evaluation of Karatusba-Ofman Multiplier Variants (in spanish)," Technical Report CINVESTAV_COMP 2006-2, 12 pages, *Computer Science Department CINVESTAVIPN*, Mexico, May 2006.

[62] D. Boneh, "On the importance of eliminating errors in cryptographic computations," *J. Cryptol.*, 2001, 14, pp. 101–119.

[63] M. Ciet, M. Joye, "Elliptic curve cryptosystems in the presence of permanent and transient faults", *Des. Codes Cryptogr.*, 2005, 36, pp. 33–43.

[64] I. Biehl, B. Meyer and V. Muller, "Differential Fault Attacks on Elliptic Curve Cryptosystems," Lecture Notes In Computer Science; Vol.1880. *Proceedings of the 20th Annual International Cryptology Conference on Advances in Cryptology*, 2000, pp.131-146.

[65] M. A. Reyhani, M. A. Hasan, "Fault detection architectures for field multiplication using polynomial bases", *IEEE Transactions on Computers*, 2006, 55, pp.1089–1103.

[66] W. Chelton, M. Benaissa, "Concurrent error detection in GF(2^m) multiplication and its application in elliptic curve cryptography," *Circuits, Devices & Systems*, IET, Vol.2, Issue:3, June 2008, pp. 289-297.

[67] C. Y. Lee, C. W. Chiou, J. M. Lin, "Concurrent error detection in a polynomial basis multiplier over GF(2^m)", *J. Electron. Test., Theory Appl.*, 2006, 22, pp. 143–150.

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

[69] C. N. Chen, S. M. Yen S, "Differential fault analysis on AES key schedule and some countermeasures," *In Proceedings of the ACISP 2003*, LNCS, Vol. 2727, pp. 118–129, 2003.

[70] P. Dusart, G. Letourneux, O. Vivolo, "Differential Fault Analysis on AES," *Cryptology ePrint Archive*: Report 2003/010 (2003).

[71] C. Giraud, "DFA on AES," *Proceedings of the AES 2004*, LNCS, Vol. 3373, pp. 27–41 (2005).

[72] D. Peacham, B. Thomas, "A DFA attack against the AES key schedule," *SiVenture*, 2006, Available on: http://www.siventure.com/pdfs/AES_KeySchedule_DFA_whitepaper.pdf (2006).

[73] M. H. Etzel, W. K. Jenkins, "Redundant Residue Number Systems for Error Detection and Correction in Digital Filters," *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. ASS-28, No 5, pp. 538-544, October 1980.

[74] S. Pontarelli, G. C. Cardarilli, M. Re, A. Salsano, "A Novel Error Detection and Correction Technigue for RNS based FIR Filters", *IEEE International Symposium*

on Defect and Fault Tolerance of VLSI Systems, 2008.

[75] P. Hämäläinen, T. Alho, M. Hännikäinen, T. D. Hämäläinen, "Design and Implementation of Low-Area and Low-Power AES Encryption Hardware Core," *in Proc. DSD,* 2006, pp.577-583.

[76] IEEE. IEEE Standard for Local and Metropolitan Area Networks—Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low- Rate Wireless Personal Area Networks (LR-WPAN), 2003. IEEE Std 802.15.4.

[77] IEEE. IEEE Standard for Local and Metropolitan Area Networks—Specific Requirements—Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications—Amendment 6: Medium Access Control (MAC) Security Enhancements, 2004. IEEE Std 802.11i.

[78] ZigBee Alliance. ZigBee Specification Version 1.0, Dec. 2004.

[79] X. Zhang, K. K. Parhi, "High-speed VLSI architectures for the AES algorithm," *IEEE Transactions on VLSI Systems*, Vol. 12, Iss. 9, pp. 957 - 967, Sept. 2004.

[80] T. Good, M. Benaissa, "AES on FPGA from the Fastest to the Smallest," *Lecture Notes in Computer Science*, Vol. 3659, pp. 427-440, Sep. 2005.

[81] P. Hämäläinen, M. Hännikäinen, and T. Hämäläinen, "Efficient hardware implementation of security processing for IEEE 802.15.4 wireless networks," in *Proc. 48th IEEE Int. Midwest Symp. on Circuits and Systems (MWSCAS 2005)*, pp. 484–487, Cincinnati, OH, USA, Aug. 7–10, 2005.

[82] P. Chodowiec, K. Gaj, "Very compact FPGA implementation of the AES algorithm," in *Proc. 5th Int. Workshop on Cryptographic Hardware and Embedded Systems (CHES 2003)*, pages 319–333, Cologne, Germany, Sept. 8–10, 2003.

[83] G. Rouvroy, F. X. Standaert, J. J. Quisquater, J. D. Legat, "Compact and efficient encryption/decryption module for FPGA implementation of the AES Rijndael very well suited for small embedded applications," *Procedings of the international conference on Information Technology: Coding and Computing 2004 (ITCC 2004)*, pp. 583 – 587, Vol. 2, April 2004.

[84] N. Pramstaller, S. Mangard, S. Dominikus, and J. Wolkerstorfer, "Efficient AES implementations on ASICs and FPGAs," *In Proc. 4th Conf. on the Advanced Encryption Standard (AES 2004)*, pp. 98–112, Bonn, Germany, May 10–12, 2005.

[85] Xilinx. Using Look-Up Tables as Shift Registers (SRL16) in Spartan-3 Generation FPGAs. Available on: www.xilinx.com/support/documentation/application_notes/xapp465.pdf (2005)

[86] Xilinx. Spartan-6 FPGA Configurable Logic Block User Guide. Available on: http://www.xilinx.com/support/documentation/user_guides/ug384.pdf

[87] E. Trichina, "Combinational logic design for AES Subbyte transformation on masked data," Available on: http://eprint.iacr.org/2003/236.pdf.

[88] A. Satoh, S. Morioka, K. Takano, S. Munetoh, "A Compact Rijndael Hardware Architecture with S-Box Optimization," *Proceedings of ASIACRYPT 2001*, LNCS Vol. 2248, pp. 239 - 254, Springer-Verlag, December 2001

[89] S. Farhan, S. Khan, H. Jamal, "Mapping of high-bit algorithm to low-bit for optimized hardware implementation," *In Proc. 16th IEEE Int. Conf. on Microelectronics (ICM 2004)*, pp. 148–151, Tunis, Tunisia, Dec. 6–8, 2004.

[90] M. Feldhofer, S. Dominikus, and J. Wolkerstorfer, "Strong authentication for RFID systems using the AES algorithm," *In Proc. 6th Int. Workshop on Cryptographic Hardware and Embedded Systems (CHES 2004)*, pp. 357–370, Boston, MA, USA, Aug. 11–13, 2004.

[91] M. Feldhofer, J. Wolkerstorfer, and V. Rijmen, "AES implementation on a

grain of sand," *IEEE Proc. Inf. Secur.*, 152(1):13–20, 2005.

[92] Y. S. Jeon, Y. J. Kim, D. H. Lee, "A Compact Memory-Free Architecture for the Aes Algorithm Using Resource Sharing Methods," *Journal of Circuits, Systems, and Computers*, Vol. 19, No. 5, pp. 1109, 2010.

[93] Helion. Tiny AES Cores. Available on: http://www.heliontech.com/aes_tiny.htm

[94] P. C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," *in Advances in Cryptology, CRYPTO '96*, LNCS 1109. Springer, 1996, pp. 104–113.

[95] P. C. Kocher, J. Jaffe, B. Jun, "Differential Power Analysis," *in Advances in Cryptology, CRYPTO '99*, LNCS 1666. Springer, 1999, pp. 388–397.

[96] K. Gandolfi, C. Mourtel, F. Oliver, "Electromagnetic analysis: concrete results," *Proc. Cryptographic Hardware and Embedded Systems: CHES 2001*, Vol. 2162 of Lecture Notes in Computer Science, pp. 251-261, Springer-Verlag, 2001.

[97] E. Biham, A. Shamir, "Differential Fault Analysis of Secret Key Cryptosystems," *in Advances in Cryptology*, CRYPTO '97, LNCS 1294. Springer, 1997, pp. 513–525.

[98] D. Boneh, R. A. DeMillo, R. J. Lipton, "On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract)," *in Advances in Cryptology, EUROCRYPT '97*, LNCS 1233. Springer, 1997, pp. 37–51.

[99] T. Malkin, F. X. Standaert, M. Yung, "A Comparative Cost/Security Analysis of Fault Attack Countermeasures," *in Fault Diagnosis and Tolerance in Cryptography, FDTC 2006*, LNCS 4236. Springer, October 2006, pp. 159–172.

[100] R. Karri, G. Kuznetsov, M. Go¨ssel, "Parity-Based Concurrent Error Detection of Substitution- Permutation Network Block Ciphers," *in the Proceedings of CHES*

2003, Lecture Notes in Computer Science, Vol. 2779, pp. 113-124, Cologne, Germany, September 2003.

[101] K. Wu, R. Karri, G. Kuznetsov, M. Goessel, "Low Cost Error Detection for the Advanced Encryption Standard," *in the Proceedings of ITC 2004*, Oct. 2004.

[102] M. Karpovsky, K. J. Kulikowski, A. Taubin, "Differential Fault Analysis Attack Resistant Architectures For The Advanced Encryption Standard," *in the Proceedings of CARDIS 2004*, Toulouse, France, August 2004.

[103] M. Karpovsky, K. J. Kulikowski, A. Taubin, "Robust Protection against Fault Injection Attacks on Smart Cards Implementing the Advanced Encryption Standard," *in the Proceedings of DSN 2004*, Florence, Italy, June 2004.

[104] N. Pramstaller, J. Wolkerstorfer, "A universal and efficient AES co-processor for field programmable logic arrays," *in Proc. FPL*, 2004, pp. 565–574.

Appendix A: The Moduli Set and Constant Values for 37-channel PRNS GF(2^{163}) Multiplier

$i \in [1, 37]$

m_i's: (in PB and binary vector format)

1	x^9+x^4+1	1000010001
2	x^9+x^5+1	1000100001
3	$x^9+x^4+x^3+x+1$	1000011011
4	$x^9+x^8+x^6+x^5+1$	1101100001
5	$x^9+x^5+x^3+x^2+1$	1000101101
6	$x^9+x^7+x^6+x^4+1$	1011010001
7	$x^9+x^5+x^4+x+1$	1000110011
8	$x^9+x^8+x^5+x^4+1$	1100110001
9	$x^9+x^6+x^4+x^3+1$	1001011001
10	$x^9+x^6+x^5+x^3+1$	1001101001
11	$x^9+x^8+x^4+x+1$	1100010011
12	$x^9+x^8+x^5+x+1$	1100100011
13	$x^9+x^7+x^2+x+1$	1010000111
14	$x^9+x^8+x^7+x^2+1$	1110000101
15	$x^9+x^7+x^4+x^2+1$	1010010101
16	$x^9+x^7+x^5+x^2+1$	1010100101
17	$x^9+x^7+x^5+x+1$	1010100011
18	$x^9+x^8+x^4+x^2+1$	1100010101
19	$x^9+x^7+x^5+x^3+x^2+x+1$	1010101111
20	$x^9+x^7+x^5+x^4+x^2+x+1$	1010110111
21	$x^9+x^8+x^7+x^5+x^4+x^2+1$	1110110101
22	$x^9+x^7+x^5+x^4+x^3+x^2+1$	1010111101
23	$x^9+x^7+x^6+x^5+x^4+x^2+1$	1011110101
24	$x^9+x^7+x^6+x^3+x^2+x+1$	1011001111
25	$x^9+x^8+x^7+x^6+x^3+x^2+1$	1111001101
26	$x^9+x^7+x^6+x^4+x^3+x+1$	1011011011
27	$x^9+x^8+x^6+x^5+x^3+x^2+1$	1101101101
28	$x^9+x^8+x^4+x^3+x^2+x+1$	1100011111
29	$x^9+x^8+x^7+x^6+x^5+x+1$	1111100011
30	$x^9+x^8+x^5+x^4+x^3+x+1$	1100111011
31	$x^9+x^8+x^6+x^5+x^4+x+1$	1101110011
32	$x^9+x^8+x^6+x^3+x^2+x+1$	1101001111
33	$x^9+x^8+x^7+x^6+x^3+x+1$	1111001011
34	$x^9+x^8+x^6+x^4+x^3+x+1$	1101011011
35	$x^9+x^8+x^6+x^5+x^3+x+1$	1101101011
36	$x^9+x^8+x^7+x^3+x^2+x+1$	1110001111
37	$x^9+x^8+x^7+x^6+x^2+x+1$	1111000111

i_i's: (in binary vector format, 9 bit each)

1	000101101
2	110111010
3	111001111
4	100010111
5	001101101
6	101010110
7	111010110
8	111010010
9	010111001
10	101100001
11	101101101
12	001001010
13	110010001
14	011101100
15	100010001
16	001111011
17	110001110
18	111001110
19	110111011
20	111111011
21	100101010
22	011101010
23	001011110
24	101111011
25	000010100
26	110011110
27	101111101
28	110000000
29	111111111
30	110101100
31	101101111
32	011111111
33	100110110
34	101000111
35	011100000
36	011011110
37	011111001

M_i's: (in Hexadecimal, 324 bits each)

1	1C70044BA86587E986055F6C6D7DEA8B9501C8FBC594880DD2CEE48B4A935815D5F6C268506F0CF109
2	1D47694BC98EB51E54956B3ECBFAE8D3D80FBA270580245E72A28D4F02CF1D25B8E1E1F584424656B9
3	1C1F293B411F8B2E4CC9B11DB008D24A9D2D206AC4C926145728AFE5FA5E70736DB3B61FB5D2A7430F
4	15AFBA6989048223BD80C22BBE85ECF6688B6F5EA55B10E2D79FD73CEFE50F4300262DBF469712B9F9
5	1D0DCA9849FDB17D5A25E7E541173104528DF3B3276CF51CB412B98C7D89E58E74FC3A3828A977E505
6	193262B49BE5913CE335960301E80D97F93E7B1826329FA61D83B24CA91B6A4757BD43031A43FDCBC9
7	1DB8954AC52EBEED1C4227426EE028DD07A0349BD3C2FDBCF0B7DB8F987274DCEAD7F3B696C3C31B07
8	167CCF07E3C1B5D7E7813658915A400C30A79F1408E54C0A722D9A441BF50E4F6D2B594FFE317A9F29
9	1FEF617A17E57ACC9D9073323FC6A39AABB3BBCD70B39E9B2641E0BDA494A2E9D67D0C47DF4459A3C1
10	1EDFA36C8DE1CB0DECE1AE7E5174120A4D01E37C63AA73B3EFF952525BE289B6C530C072AACF090071
11	17D5C9F199B3B495895E58F2C6DD4362771A019951C902E9B385129C4624ECF8272BB7766F44FC5FA7
12	16B25C9F591B29D9139920FAE13C9FF9E5CF151B8F447AB97057567AD5091A71C742725892294C2FD7
13	1A326984B399993D876522593325077F68732226B2440ABC2A480A520867EA3D75EC5B1DF288B219B3
14	10F030193EC04CE3BB1111753BD5224994DBA6696D9F72AA225DEC983AD5D5961E00D3C6BBE5F810AD
15	1AC3ACAB5EBB8C8D5FE10E41400098BD00D00D737D963F420CF4FF79B952BDC2AA7E318E0E27D4C43D
16	1BE1EB36F4284BE14ADC7868E94231000F59A90D7145EB4D13D7010C098E3CEACE2265CCBD6D930C8D
17	1BC666EA7584BAE5959536693805856387B02561DDC4A24D505A89F386F5119E51BEAFE52981475457
18	17FBB27966D376A4077F76938CEFB42EDF8A83FEE2AD031B6CD0D2FBFF12DE6EB43FE0328B773589BD
19	1B8850CE939F946F3910AD9366516945DD0D6FE7B5BCFED6177452D14CD936A1139EC53F392E61423B
20	1B10A275FAC87E376BE72D28B6C855557BC0DA872C1E4ED2B734E235845C7B7BD66C80E886EE1356C3
21	118318CBDA7F1A7BAFFE8CBA5E376FDA323DAB476898ABE38D1DF76A6C3E480A1A4BC943533727B81D
22	1B790D3416ED64BDB6FC281B54B329E3EF955AE88E22AC12D205E3A8997BA070D5BD8AB2DD1B321015
23	18C2930EAAED781C92CBE00BC05AA032B552332088E29AEEEF8FB0C7F82198EAC3F60B5BD16623E35D
24	198FD9C28E47500F3C440AEF7199FBAAEA5D1F9DA514BA811860CC47B9DF401DE988DDBC53DB0D4EDB
25	13B8829ED86EC0850375D4F3A6504F08269D362EB685BE1A3D667C5A5250654AF095BF9F0E3768B7E5
26	195A3BFD221DDFBE115CACAA6A5902663909E694EBA86EFC9B6415FBE6A52F2BA05CC08EC974689ECF
27	15F3CFCEC6DE1F9EFA510F08E91FAC6BFECA5F3E64B49738204968F38818690CF7A5B68ACFFD72AE45
28	1788F9D8A3600E7378B88767FD7AD8F2D50CFE3785CA3854EE29A1B8431CD960BA10585615F0B6AACB
29	1257184279A1146DBA8D0AB93A9050786F5C8A883EB31249F3F9C73CEB5884428F3F42150D84A20217
30	160FF211EAD6C6A60282FA39CF2A6F8FA9B41117BCB8CDC55A6CDBBD57FE4660CF36305858EC852FAF
31	15621782AE6302A8032CA43384E26197588CD6B6F690C980BD7EAAF9374E2680C24A6FD2A48A2C0D47
32	145E90801DE1457C556DA667886109263 2D741FCAC91B7C83E85084E66D2FE2E46E99817346BEF0D5B
33	1394291A92FC5AA61BF7043AE02A90E735416543F149A1136DED2443170334CA2FD3BD91BE12B0261F
34	14BD4E0DD00DA8E1ECC15C89C77753A549745F90B5AD4DF92423CB4956CD638F15BFE285B7DC4D054F
35	15DD0CFCAA76D0A1B97515E3B12CB48AEDCC154A714D9E00C8CDF1D923E8C08280560D8B1B429F313F
36	1085E9B405395419FFF31FE08FD69A3774C44CCDEC3B153BB1FD215475ADA7CA59AF709A3C5271FC9B
37	13CCC4AC828619B1B5D6FB591142132FA4B92BD44C3F15B8F74EAAE2A8A28BCC64F5C2475F3C14FBF3

Appendix B: The Moduli Set and Constant Values for 4-channel PRNS GF(2^{163}) Multiplier

$$p(x) = \sum_{i=1}^{4} (p_i(x) \cdot I_i(x) \bmod m_i(x)) \cdot M_i(x)$$

$$\text{Where } M_i(x) = \frac{M(x)}{m_i(x)} = m_1(x) \dots \cdot m_{i-1}(x) \cdot m_{i+1}(x) \dots \cdot m_4(x)$$

$$I_i = M_i^{-1}(x) \big(\bmod m_i(x)\big) \text{ for } i \in [1,4]$$

$$m_1 = x^{84} + x^5 + 1$$
$$m_2 = x^{84} + x^9 + 1$$
$$m_3 = x^{84} + x^{11} + 1$$
$$m_4 = x^{84} + x^{13} + 1$$

$$M_1 = m_2(x) \cdot m_3(x) \cdot m_4(x) = x^{252} + x^{181} + x^{179} + x^{177} + x^{168} + x^{108} + x^{106} + x^{104} + x^{84} + x^{33} + x^{24} + x^{22} + x^{20} + x^{13} + x^{11} + x^9 + 1$$

$$M_2 = m_1(x) \cdot m_3(x) \cdot m_4(x) = x^{252} + x^{181} + x^{197} + x^{173} + x^{168} + x^{108} + x^{102} + x^{100} + x^{84} + x^{29} + x^{24} + x^{18} + x^{16} + x^{13} + x^{11} + x^5 + 1$$

$$M_3 = m_1(x) \cdot m_2(x) \cdot m_4(x) = x^{252} + x^{181} + x^{177} + x^{173} + x^{168} + x^{106} + x^{102} + x^{98} + x^{84} + x^{27} + x^{22} + x^{18} + x^{14} + x^{13} + x^9 + x^5 + 1$$

$$M_4 = m_1(x) \cdot m_2(x) \cdot m_3(x) = x^{252} + x^{179} + x^{177} + x^{173} + x^{168} + x^{104} + x^{100} + x^{98} + x^{84} + x^{25} + x^{21} + x^{16} + x^{14} + x^{11} + x^9 + x^5 + 1$$

$I_1 = M_1^{-1}(x)\big(\bmod m_1(x)\big) =$
1000011101111111011000101101100001110110100010010011001110011101011000101101100101111

$I_2 = M_2^{-1}(x)\big(\bmod m_2(x)\big) =$
1101010011010110011111011000001011010111100000101101011110000010110101111001100000100

$I_3 = M_3^{-1}(x)\big(\bmod m_3(x)\big) =$
0100101011011100000000110011111111011101010101110111111111011101010101110110110101101

$I_4 = M_4^{-1}(x)\big(\bmod m_4(x)\big) =$
0001100101110101000110010010010100011011101001010001101110100101000110111001011111111

Appendix C: The Moduli Set and Constant Values for Fault Tolerant 5-channel RPRNS GF(2^{163}) Multiplier

$m_1 = x^{127} + x + 1$
$m_2 = x^{127} + x^7 + 1$
$m_3 = x^{127} + x^{15} + 1$
$m_4 = x^{127} + x^{30} + 1$
$m_5 = x^{127} + x^{63} + 1$

SRC 1 (m_2, m_3, m_4, m_5)
$M_1 = (x^{127} + x^{15} + 1)(x^{127} + x^{30} + 1)(x^{127} + x^{63} + 1)$
$M_2 = (x^{127} + x^7 + 1)(x^{127} + x^{30} + 1)(x^{127} + x^{63} + 1)$
$M_3 = (x^{127} + x^7 + 1)(x^{127} + x^{15} + 1)(x^{127} + x^{63} + 1)$
$M_4 = (x^{127} + x^7 + 1)(x^{127} + x^{15} + 1)(x^{127} + x^{30} + 1)$
I_1 = 1000110010101100011111100110001100000100010001001100011101111001111110100001 1111000010110100000100001110001101000100000111 0000
I_2 = 00011001000111011100011000111011100011000111000011100000111001100011100111000 10001111011100011110000111100001001111011110001011
I_3 = 1000100000011010001101111110100101000010010110111101010011111000000010001011 001111111011001001110110110001100110011100000011111
I_4 = 0001110110101011100011111011000111001010011011111111001101100111110101101100 0 0111000101010111001011010100001011101001110110010 1

SRC 2 (m_1, m_3, m_4, m_5)
$M_1 = (x^{127} + x^{15} + 1)(x^{127} + x^{30} + 1)(x^{127} + x^{63} + 1)$
$M_2 = (x^{127} + x + 1)(x^{127} + x^{30} + 1)(x^{127} + x^{63} + 1)$
$M_3 = (x^{127} + x + 1)(x^{127} + x^{15} + 1)(x^{127} + x^{63} + 1)$
$M_4 = (x^{127} + x + 1)(x^{127} + x^{15} + 1)(x^{127} + x^{30} + 1)$
I_1 = 0010110111100110011110010100101011111001011010101011110101101110101011101100 0 0101001100011100011001011110110000100011011101111 0
I_2 = 0101111110111001101001010001101100100011100100000000111100100011010100110111 0 0111001001000110100000001000110100110111001101101111
I_3 = 1010100000011111111100011001101000011011111001011000111111101110100000000101000 0101111010111011110000100011001010101111010111 1001
I_4 = 1101101001100000000111111011001011110110100110001101111001111010111010011000 00 1110111011000110000100101100111110010010101010 0011

SRC 3 (m_1, m_2, m_4, m_5)
$M_1 = (x^{127} + x^7 + 1)(x^{127} + x^{30} + 1)(x^{127} + x^{63} + 1)$
$M_2 = (x^{127} + x + 1)(x^{127} + x^{30} + 1)(x^{127} + x^{63} + 1)$
$M_3 = (x^{127} + x + 1)(x^{127} + x^7 + 1)(x^{127} + x^{63} + 1)$
$M_4 = (x^{127} + x + 1)(x^{127} + x^7 + 1)(x^{127} + x^{30} + 1)$
I_1 = 0111101100100010010110100111010110010111000001001101011100000111011001011011 0101001111011011101000000101101100001111101101101 11
I_2 = 0110000101010111010000000110011011011011111011000000111010111001000000011001 1100010101000111011001000100001010101110011000100 00
I_3 = 0001001100100010100111110110110110110100100111011010101100100010000000101101 0 1110101100110101000011111010110000011110111000101 10
I_4 = 0000100101010111100001010111111011111000011101010111001010011100011010001101 0000010111111110001001001010101111111110011100010100

SRC 4 (m_1, m_2, m_3, m_5)

$M_1 = (x^{127} + x^7 + 1)(x^{127} + x^{15} + 1)(x^{127} + x^{63} + 1)$

$M_2 = (x^{127} + x + 1)(x^{127} + x^{15} + 1)(x^{127} + x^{63} + 1)$

$M_3 = (x^{127} + x + 1)(x^{127} + x^7 + 1)(x^{127} + x^{63} + 1)$

$M_4 = (x^{127} + x + 1)(x^{127} + x^7 + 1)(x^{127} + x^{15} + 1)$

$I_1 =$ 1100001011000000001011000000101110110010011110110010011100100101010111100101
0101111011010101101100111110001100110110001101101 01

$I_2 =$ 0001100101001011011100011100011110100110001000010100110110001101000111100010
0100100110101101101111011110101110100101101001010 01

$I_3 =$ 1010101010011110101010100111100101010101110011010101011100111101010101001111
0010101010111101110101000111011011000101000010011 1

$I_4 =$0111000100010101111101111011010101000001101111001100000100110110100110110001 1
1011101010111011011000000101000001101111010110010 0

SRC 5 (m_1, m_2, m_3, m_4)

$M_1 = (x^{127} + x^7 + 1)(x^{127} + x^{15} + 1)(x^{127} + x^{30} + 1)$

$M_2 = (x^{127} + x + 1)(x^{127} + x^{15} + 1)(x^{127} + x^{30} + 1)$

$M_3 = (x^{127} + x + 1)(x^{127} + x^7 + 1)(x^{127} + x^{30} + 1)$

$M_4 = (x^{127} + x + 1)(x^{127} + x^7 + 1)(x^{127} + x^{15} + 1)$

$I_1 =$1100100100111101101011011100101011100011010001111101010010111100111101011000
0 0010101101000011100010000111010010001101010010 0101

$I_2 =$0010100111110001101001110111101100100111100100010011000110100000110111011101
1110111111011000100010101000000011001110111011 0111

$I_3 =$ 1010101001010101010101101010101010101011010101010101011010101110101010010100 1
0010100011011000110110110001101100011001010011101011

$I_4 =$0100101010011001010111000001101100000111111101101100001001101110011001111101 0
0110011111111000011010100011000110100110000100000 0