High performance hardware security components

Wang, Yi

2008

Wang, Y. (2008). High performance hardware security components. Doctoral thesis,
Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/2603>

<https://doi.org/10.32657/10356/2603>

Nanyang Technological University

Downloaded on 26 Apr 2023 18:47:17 SGT

**NANYANG
TECHNOLOGICAL
UNIVERSITY**

HIGH PERFORMANCE HARDWARE
SECURITY COMPONENTS

**HIGH PERFORMANCE HARDWARE
SECURITY COMPONENTS**

WANG YI

WANG YI

SCHOOL OF COMPUTER ENGINEERING

2008

2008

High Performance Hardware Security Components

Wang Yi

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in fulfilment of the requirement for the degree of
Doctor of Philosophy

2008

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Associate Professor Douglas L. Maskell, for his supervision and academic motivation throughout this research programme. Although, it is only very short period for me to study under him, but his intangible skills and philosophical aspects of research influenced me a lot in my future research career.

I would also like to thank Doctor Jussipekka Leiwo, for his guidance and proposal in this research work.

I would also like to thank Professor Thambipillai Srikanthan, for providing me the research suggestions and research environment in CHIPES to support me fulfill my research.

I would like to thank Tham Kwang Shuen and Wang Lingyu for their friendship, help, discussions, suggestions and collaboration pertaining to the work in this research programme.

To the staff of Centre for High Performance Embedded Systems, I wish to convey my appreciation to all of them for their kind and friendly assistance.

Last, I would like to express my gratitude to my parents for their support and encouragement all the time. I am grateful to my husband, Xiang Zhitao (向志涛), for his love and unwavering support. This dissertation is specially dedicated to him.

ABSTRACT

A considerable amount of recent research has focused on hardware implementations of cryptographic algorithms and protocols because of the higher performance and the better security they provide, compared to software-based implementations. Software-based implementations are more appealing in that they can support a range of different security protocols, whereas hardware implementations are generally developed for a specific protocol. To improve the flexibility of hardware implementations, it would be desirable to be able to support multiple security protocols on the same platform. Thus it would be useful to design a security coprocessor module which is able to unify the arithmetic computations for supporting protocols like Rivest, Shamir, and Adleman (RSA) [Rive78] and Elliptic Curve Cryptosystem (ECC) [Kobl87, Mill85] over Galois fields $GF(p)$ and $GF(2^p)$.

Additionally, while specialized hardware security components are perceived to provide better security, their widespread use has presented new and different opportunities for cryptanalysis attacks. Rather than trying to break the cryptographic algorithm, attackers are focusing on attacking deficiencies in the hardware implementation. These attacks, called side channel attacks, are a powerful technique for determining sensitive data based on timing measurements, power analysis, fault analysis, etc. Hence, the ability to defend against cryptanalysis of the hardware must also be considered.

In this research, we have proposed a unified architecture for a security coprocessor based on the analysis of basic cryptographic algorithms and protocols. Because of the importance of modular multiplication in security applications, we have developed a high performance modular multiplication algorithm which is able to reduce the total computational time by nearly half compared to other state-of-the-art algorithms. As this can be extended to modular exponentiation, it provides an efficient method for computing modular operations over $GF(p)$.

We then examine the current state of unified architectures, and propose and design a new signed digit adder that provides a better balance between the carry propagation modes and the carry-less modes of operation. Based on this adder module, we develop a dual-field arithmetic unit which is able to support modular multiplication, modular addition/ subtraction and modular inversion over $GF(p)$ and $GF(2^p)$. This module provides better performance than other designs in the research literature.

Side channel attacks, and in particular power analysis attacks, have resulted in the development of a number of countermeasures for hardware based security components. We develop an “initial value DPA” attack and show that it is an effective attack against state-of-the-art power analysis attack countermeasures. We then go on to develop our own, more robust, countermeasure to protect against power analysis attacks for both RSA and ECC implementations.

Finally, we propose a framework for a unified security coprocessor based on the algorithms and computational modules developed in this thesis.

CONTENTS

ACKNOWLEDGEMENTS	i
ABSTRACT	ii
CONTENTS	iv
LIST OF ALGORITHMS	viii
LIST OF FIGURES	x
LIST OF PROTOCOLS	xii
LIST OF TABLES	xiii
LIST OF ABBREVIATIONS	xiv

CHAPTER 1

INTRODUCTION	1
---------------------	----------

1.1	Motivation	1
1.2	Objective	2
1.3	Thesis Contribution	3
1.4	Thesis Organization	5

CHAPTER 2

BACKGROUND AND LITERATURE REVIEW	7
---	----------

2.1	Public Key Cryptosystems	7
2.1.1	RSA Cryptosystems	8
2.1.1.1	The Definition of the RSA Algorithm	8
2.1.1.2	Modular Exponentiation	9
2.1.1.3	The State of Art for RSA Hardware Implementation	10
2.1.2	Elliptic Curve Cryptosystems (ECC)	12
2.1.2.1	A Curve Defined over $GF(p)$	13
2.1.2.2	A Curve Defined over $GF(2p)$	13
2.1.2.3	The State of the Art for ECC Hardware Implementation	14
2.1.3	Discrete Logarithm Based Cryptosystems	15
2.1.3.1	ElGamal encryption and the ElGamal signature scheme	15
2.1.3.2	Signcryption	16
2.2	Cryptographic Coprocessors	17
2.2.1	General Commercial Crypto-coprocessors	17

2.2.2	RSA based Cryptographic Coprocessors	20
2.2.3	ECC based Cryptographic Coprocessors	21
2.2.4	A Unified Architecture for Cryptographic Coprocessor	23
2.3	Side Channel Attacks	25
2.3.1	Power Analysis	25
2.3.2	Countermeasures against Power Analysis Attacks	27
2.4	Conclusion	30

CHAPTER 3

AN EFFICIENT MODULAR MULTIPLICATION ALGORITHM 31

3.1	Introduction	31
3.2	Montgomery Multiplication	31
3.3	Blum's modified Montgomery modular multiplications	34
3.4	An Improved Modular Multiplication Algorithm	35
3.5	Hardware Implementation	37
3.6	Performance Evaluation	41
3.6.1	Software Performance of our Proposed Algorithm	41
3.6.2	Hardware Performance of our Proposed Algorithm	42
3.7	Conclusions	44

CHAPTER 4

AN SD ADDER BASED UNIFIED MULTIPLIER FOR FINITE FIELDS $GF(p)$ AND $GF(2^p)$ 45

4.1	Introduction	45
4.2	Preliminaries	46
4.2.1	The SD Number System	46
4.3	Algorithms for Implementing RSA and ECC	47
4.3.1	Proposed Modular multiplication over $GF(p)$ and $GF(2^p)$	47
4.3.2	Proposed Modular Addition and Subtraction over $GF(p)$ and $GF(2^p)$	51
4.3.3	Proposed Modular Inversion over $GF(p)$ and $GF(2^p)$	52
4.4	Hardware Implementation	54
4.4.1	A Public Key Cryptographic Coprocessor	54
4.4.2	Unified Adder over $GF(p)$ and $GF(2^p)$	55
4.4.3	A Unified Modular Multiplier over $GF(p)$ and $GF(2^p)$	56
4.4.4	Unified Modular Inversion over $GF(p)$ and $GF(2^p)$	57

4.4.5	Special Memory	58
4.4.6	Performance Evaluation	59
4.5	Comparisons and Results	59
4.6	Conclusion	63

CHAPTER 5

COUNTERMEASURES FOR POWER ANALYSIS ATTACKS 65

5.1	Introduction	65
5.2	Power Attack Countermeasures for ECC	66
5.2.1	Mamiya's Countermeasure for ECC	66
5.2.2	Kim's Countermeasure	67
5.2.3	The weakness of Kim's Countermeasure for ECC	67
5.2.4	Proposed Countermeasure against DPA for ECC	68
5.3	Power Attack Countermeasures for RSA	69
5.3.1	Mamiya's Countermeasure	70
5.3.2	Kim's Countermeasure	71
5.3.3	Weakness of Kim's Countermeasure for RSA	71
5.3.4	Proposed Countermeasure against DPA for RSA	72
5.4	Evaluation of the Proposed Countermeasures	73
5.5	Conclusion	77

CHAPTER 6

A DPA-RESISTANT UNIFIED ARCHITECTURE FOR PUBLIC KEY CRYPTOSYSTEMS 79

6.1	Introduction	79
6.2	Unified Architecture for Signcryption	79
6.3	DPA-resistant Modular Exponentiation	79
6.4	The Proposed Crypto-processor	81
6.4.1	Unified Execution Unit	83
6.4.2	Modular Multiplication	84
6.4.3	Pseudo Prime Number Generator	86
6.4.4	The Enveloping Unit	87
6.5	RSA Calculations	88
6.6	Conclusion	90

CHAPTER 7	
CONCLUSION AND FUTURE WORK	91
7.1	Conclusions 91
7.1.1	High Performance Modular Multiplication 91
7.1.2	Unified Modular Arithmetic Unit over $GF(p)$ and $GF(2p)$ 91
7.1.3	Countermeasures against Power Attacks 92
7.1.4	A DPA-resistant Unified Cryptographic Coprocessor 92
7.2	Future Work 92
7.2.1	Supporting Private Key Cryptosystems 92
7.2.2	Application Specific Instruction-set Processor 93
7.2.3	Defending Against Side Channel Attacks (Timing and Fault Attacks) 93
7.2.4	Develop a Prototype of the Proposed Architecture 93
BIBLIOGRAPHY	95
APPENDIX	103
A.	Elliptic Curve Arithmetic 103
A.1	General Form of EC 103
A.2	Homogeneous Projective Coordinate 103
A.3	Jacobian Projective Coordinate 104
A.4	Montgomery Form 104
A.5	Hessian Form 105
B.	Signature, Encryption and Signcryption Schemes 106

LIST OF ALGORITHMS

Algorithm 2.1	Key generation for RSA public-key encryption [Mene97]	8
Algorithm 2.2	RSA public-key encryption [Mene97]	8
Algorithm 2.3	Basic modular exponentiation	9
Algorithm 2.4	Montgomery modular multiplication [Mene97]	9
Algorithm 2.5	The binary method for computing $Q = eP$	28
Algorithm 2.6	Coron's first countermeasure: Randomizing the exponent [Coro99]	28
Algorithm 2.7	Coron's second countermeasure: Blinding the point P [Coro99]	28
Algorithm 2.8	Coron's third countermeasure: Randomized projective coordinates [Coro99]	29
Algorithm 2.9	Binary expansion with a random initial point	29
Algorithm 2.10	Kim's exponentiation algorithm which is immune to DPA, SPA and timing attacks [Kim04b]	30
Algorithm 3.1	Our proposed modular exponentiation against linear timing attacks	32
Algorithm 3.2	The radix-2 Montgomery multiplication algorithm [Eldr93]	33
Algorithm 3.3	High radix Montgomery multiplication algorithm [Orup95]	33
Algorithm 3.4	Blum's radix-2 Montgomery modular multiplication [Blum01]	34
Algorithm 3.5	Blum's modified high-radix Montgomery modular multiplication [Blum01]	35
Algorithm 3.6	The proposed high-radix Montgomery modular multiplication	35
Algorithm 3.7	Proposed high-radix Montgomery modular multiplication for hardware implementation	37
Algorithm 4.1	A modified Montgomery's modular multiplication algorithm [Eldr93]	48
Algorithm 4.2	The proposed modular multiplication algorithm over $GF(p)$ based on an SD adder	49
Algorithm 4.3	The proposed modular multiplication algorithm over $GF(2^p)$ based on an SD adder	50

Algorithm 4.4	The proposed modular addition and modular subtraction over $GF(p)$ based on an SD adder	51
Algorithm 4.5	The proposed modular addition and modular subtraction over $GF(2^p)$ based on an SD adder	52
Algorithm 4.6	The proposed modular inversion algorithm over $GF(p)$ based on an SD adder	53
Algorithm 4.7	The proposed modular inversion algorithm over $GF(2^p)$ based on an SD adder	53
Algorithm 5.1	Binary method (MSB) [Sato98]	66
Algorithm 5.2	Montgomery's method [Mont87]	66
Algorithm 5.3	Mamiya's method to compute eP for ECC	67
Algorithm 5.4	Kim's method to compute eP for ECC	67
Algorithm 5.5	Our proposed method to compute eP for ECC	69
Algorithm 5.6	The proposed countermeasure against SPA for RSA	70
Algorithm 5.7	Mamiya's method to compute $d^e \bmod M$ for RSA	70
Algorithm 5.8	Kim's method to compute d^e for RSA	71
Algorithm 5.9	Our proposed method to compute d^e for RSA	72
Algorithm 5.10	Our proposed method to compute eP for ECC in a general computation form	73
Algorithm 6.1	The proposed algorithm to compute $s(x) = a(x) \cdot b(x) \cdot x^{-n} \bmod m(x)$ based on a signed-digit adder	85
Algorithm 6.2	the proposed algorithm to compute $A \cdot B \cdot R^{-1} \bmod M$ based on a signed-digit adder	86
Algorithm 6.3	Produce a private key	89
Algorithm B.1	The general function of unifying signing, encryption and signcryption	106

LIST OF FIGURES

Figure 2.1	Systolic array with processing elements (PE)	10
Figure 2.2	The block diagram of SLE88CFX4000P, 32-bit CPU and peripherals [SCIC04]	18
Figure 2.3	Providing security against malicious software attack TrustZone [Alve04]	18
Figure 2.4	The components of an embedded system-on-chip architecture demarcated into secure and non-secure areas [Alve04]	18
Figure 2.5	The IBM 4758 cryptographic coprocessor schematic [IBM02]	20
Figure 2.6	The block diagram of a private and public key crypto-processor [Kim04a]	22
Figure 2.7	Architecture for Sutikno's arithmetic processor [Suti98b]	22
Figure 2.8	The structure of (a) a full adder and (b) a dual-field adder [Sava00]	24
Figure 2.9	The architecture for a dual-field arithmetic unit [Wolk02]	24
Figure 3.1	The high speed iterative-add architecture for modular multiplication	39
Figure 3.2	The internal structure of a processing element for the high speed iterative-add architecture	39
Figure 3.3	The data flow diagram showing the flow of data through the proposed iterative-add architecture	40
Figure 3.4	The run time (in <i>ns</i>) for (a) Blum's algorithm and (b) our proposed algorithm on a Pentium 4 (2.4 GHz) processor	41
Figure 3.5	The run time (in <i>ns</i>) for (a) Blum's algorithm and (b) the proposed algorithm on FPGA	42
Figure 3.6	The implementation's total dynamic power consumption (mW) for different key lengths on a 0.18 μ m CMOS ASIC	44
Figure 4.1	The structure of a full adder and an SD adder	46
Figure 4.2	The structure for RSA and ECC operations	55
Figure 4.3	Dual-field modular addition and subtraction	56
Figure 4.4	The implementation of dual-field modular multiplication	57
Figure 4.5	The implementation of the dual-field SD adder-shifter	58
Figure 4.6	A fast dual-field modular multiplication unit	60
Figure 5.1	The computation wave form of (a) Kim's algorithm and (b) the proposed algorithm	76

Figure 5.2	The power consumption waveform of (a) Kim's algorithm and (b) the proposed algorithm	77
Figure 6.1	The overall architecture	80
Figure 6.2	The proposed architecture for a crypto-processor	82
Figure 6.3	The unified execution unit for computing over $GF(p)$ and $GF(2^p)$	82
Figure 6.4	The format of the crypto-processor instructions	82
Figure 6.5	The structure of $MODM(A, B, M, k, n)$	85
Figure 6.6	The process of encryption and decryption of private keys for transfer between the execution unit and memory	87
Figure 6.7	The flowchart (a) for setting up the RSA keys and (b) for performing modular exponentiation	89

LIST OF PROTOCOLS

Protocol 2.1	Diffie-Hellman key agreement (basic version)	15
Protocol 2.2	ElGamal public-key encryption scheme	15
Protocol 2.3	Encryption scheme [An02]	16
Protocol 2.4	Signature scheme [An02]	16
Protocol 2.5	Signcryption scheme [An02]	16

LIST OF TABLES

Table 2.1	The relationship between power attacks and the common cryptographic algorithms	27
Table 3.1	Modular exponentiation of hardware performance comparison	43
Table 4.1	Rules for generating the carry (c_{i+1}) and sum (u_i) results for different inputs x_i y_i in the SD number system	47
Table 4.2	Rules for computing s_i from the intermediate result u_i and the carry c_i in the SD number system	47
Table 4.3	The performance of the proposed design on Virtex-4 XC4VFX100, Virtex-E XCV2000E and Virtex XC40150XV FPGA devices	60
Table 4.4	The performances of the proposed design for different timing constraints on 0.09 μm CMOS standard cell technology	61
Table 4.5	Multiplier hardware performance comparison	61
Table 4.6	Unified hardware performance comparison	62
Table 5.1	Computation cost of ECADD and ECDBL for different EC forms	74
Table 5.2	Computation cost of M, S and I	74
Table 5.3	The computational cost of a 160-bit, 5 point elliptic curve computation	74
Table 6.1	Procedures for signature, encryption and signcryption schemes	80
Table 6.2	The codes for the key length and radix value	83
Table 6.3	The equations for determining primality	87
Table 6.4	Definition and life time of different keys	88

LIST OF ABBREVIATIONS

AES	Advanced Encryption Standard
ASIC	Application-Specific Integrated Circuit
ASIP	Application Specific Instruction-set Processor
BRIP	Binary expansion with a Random Initial Point
CIHS	Coarsely Integrated Hybrid Scanning
CLA	Carry Look-ahead Adder
CMOS	Complementary Metal-Oxide-Semiconductor
CRT	Chinese Remainder Theorem
CPA	Carry Propagate Adder
CSA	Carry Save Adder
DES	Data Encryption Standard
DPA	Differential Power Analysis
EC	Elliptic Curve
ECC	Elliptic Curve Cryptosystem
EEA	Extended Euclidean Algorithm
FA	Full Adder
FPGA	Field Programmable Gate Array
HA	Half Adder
HW/SW	Hardware/Software
IP	Intellectual Property
LSB	Least Significant Bit
LUT	Lookup Table
MESD	Multiple-Exponent, Single-Data
MSB	Most Significant Bit
NLFSR	Non-linear feedback shift register
OEM	Original Equipment Manufacturer
PCI	Peripheral Component Interconnect
PE	Processing Element
PKC	Public Key Cryptography
PKCS	Public Key Cryptography Standard
RISC	Reduced Instruction Set Computing
RNS	Residue Number System

RPA	Refined Power Analysis
RSA	Rivest, Shamir, and Adleman
RSD	Redundant Signed Digit
SD	Signed Digit
SEMD	Single-Exponent, Multiple-Data
SHA-1	Secure Hash Algorithm 1
SOC	System On Chip
SPA	Simple Power Analysis
T-DES	Triple – Data Encryption Standard
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLSI	Very Large Scale Integration
XOR	Exclusive OR
ZEMD	Zero-Exponent, Multiple-Data
ZPA	Zero Power Analysis

Chapter 1 Introduction

1.1 Motivation

Cryptology can be defined as the science concerned with communication in secure and usually secret form. It encompasses both cryptography and cryptanalysis. The former involves the study and application of principles and techniques by which information is rendered unintelligible to all but the intended receiver, while the latter is the science and art of solving cryptosystems to recover such information [Piep03].

Today this definition needs to be extended as modern cryptology focuses its attention on the design and evaluation of a wide range of methods and techniques for information protection. Information protection covers not only secrecy but also authentication, integrity, verifiability, non-repudiation and other security objectives.

With the development of more and more cryptographic algorithms and protocols and their implementation into hardware, a number of techniques, called side channel attacks, have been developed and implemented against these implementations. Side channel information is information that can be retrieved from the encryption device that is neither the plaintext to be encrypted nor the ciphertext resulting from the encryption process. Therefore, defending against side channel attacks should be taken into account when designing a secure system.

Thus our focus is on how to improve the security of a system and whether the resulting security implementation should be implemented in hardware or software. A common approach is to have dedicated hardware that manages the most critical security processing and supporting software that handles the control of the security processing. Adding more complex security processing in hardware increases the security and performance but also increases the complexity of hardware and complicates the interaction with software. A balance must be found between complexity, performance and security.

Software based security cannot provide a secure environment because the important information is not directly controlled by the user. A typical case would be when a user wants to release some unused temporary keys during protocol execution but does not have access rights to the physical memory location that stores the keys. This may lead to possible attacks because the processor will maintain the key at the original location until it is replaced. Therefore, to avoid such a problem, control over the memory

management must be granted to the user and provision must be made to separate secure data from non-secure data.

A significant amount of current development focuses on improving the security of hardware-based implementations. The difficulties of designing a security coprocessor are complexity, large data transformation and cost. The faster development cycle of FPGA technology makes it possible to design a security coprocessor to meet this demand. Furthermore, some attacks aim to compromise a hardware implementation using timing measurements, power analysis and fault analysis, etc. Therefore, to design a secure system, based on a modified algorithm capable of resisting side channel attacks, is a desirable method to avoid these weaknesses.

1.2 Objective

The objective of this thesis is to explore a unified architecture for public key cryptosystems to support the basic computations of Rivest-Shamir-Adleman (RSA) and Elliptic Curve Cryptogram (ECC) on Galois fields $GF(p)$ and $GF(2^p)$. The research involves the proposal of a framework for a new unified cryptographic coprocessor for executing a group of cryptographic algorithms and protocols with similar characteristics, and extracting the basic operations and arithmetic units to support those algorithms and protocols. The cryptographic coprocessor includes a unified execution unit, a controller, and other supporting components. The unified execution unit consists of a pseudo random number generator, a large prime random number generator, an enveloping unit and a modular multiplication unit. We propose a unified modular multiplier to support the basic computations of ECC, and RSA Cryptogram on $GF(p)$ and $GF(2^p)$. We also explore a novel countermeasure against power analysis attacks for operations with sensitive data as the exponent. The study also involves an investigation of the balance between performance, power, delay and area constraints for the coprocessor. Hardware metrics in terms of area, delay and power consumption of elements of the proposed cryptographic coprocessor are also analyzed and evaluated. Thus, given the main objective, the following scope has been covered in this thesis.

- 1) The proposal of an efficient modified Montgomery's modular multiplication for speeding up the computation of modular exponentiation. An evaluation of the modular multiplier with area, delay and power consumption at different radix values and key sizes is performed.
- 2) The proposal of a novel unified architecture for public key cryptosystems based on the signed digit number system to achieve a tradeoff between carry propagation operation and carry-less modes of operation.

- 3) The proposal of a new countermeasure against power analysis attacks without increasing the computational complexity.
- 4) The proposal of an overall framework for a public key crypto-processor including a DPA-resistant unified execution unit, signed digit data converter, controller, memory and a bus interface.

It is hoped that based on the results of our research, a larger design space is formulated from which a designer could build a cryptographic coprocessor with the ability to defend against side channel attacks without reducing system performance.

1.3 Thesis Contribution

The main contributions of this thesis are:

1. We propose a modified Montgomery modular multiplication algorithm to reduce the total computation time compared to Blum's algorithm [Blum01] and extended it to modular exponentiation. The number of loop iterations is reduced by half, by using the consecutive bits a_{n-1} , a_n , a_{n+1} instead of the original a_n for each computation step. Thus, we are able to achieve higher performance by trading timing and area without increasing the computational complexity. The simulation results have revealed that our proposed algorithm is faster and consumes less power than existing methods.
2. Much of the previous work on developing a unified architecture for public key cryptosystems on $GF(p)$ and $GF(2^p)$ aims to shorten the computation time. Instead of using carry save adders, we develop an alternative method to balance the carry propagation operations. This method uses an efficient modified adder based on the signed digit (SD) number representation. We propose a unified architecture for supporting the operations of RSA and ECC on $GF(p)$ and $GF(2^p)$. The supported operations of our unified architecture include modular exponentiation (RSA), point adding (ECC) and point doubling (ECC). The dual-field arithmetic unit has been applied to implement modular addition/subtraction, modular multiplication and modular inversion in both fields. A pipelining methodology is also used to implement the proposed design to reduce computation time. The critical path of the proposed design includes only two levels of Look Up Table (LUT) and some selection logic. The simulation results have shown that the proposed architecture achieves better performance than other full adder based designs.
3. We propose a DPA resistant modular exponentiation algorithm which has minimal effect on the computational complexity. Several techniques in the literature have been proposed to increase the resistance of RSA against SPA

and DPA. We have addressed one possible attack, called an "initial value DPA" attack, to successfully attack Kim's countermeasures against power analysis when computing modular exponentiation [Kim05]. We then propose a refined countermeasure using a variable initial value instead of the constant initial value. Finally, dynamic power analysis of the proposed algorithm demonstrates that it is superior to Kim's techniques for minimizing power variations, thus is less susceptible to power attacks.

4. Security is an emerging concern in the design of embedded systems. Various side channel attacks must be tackled by the implementation of a secure embedded system. We propose a framework for a unified cryptographic coprocessor, based on our above work, that can resist probing attacks on the internal bus and memory, as well as SPA, DPA, ZPA and RPA. Private keys are maintained within the execution unit to reduce the possibility of leakage during cryptographic operations. The overall architecture includes a DPA-resistant unified execution unit, a signed digit data converter, controller, memory and bus interface. The scalable arithmetic unit supports different key sizes and achieves high performance.

Publications arising from this work

1. [Wang07a] Yi Wang, Douglas L. Maskell and Jussipekka Leiwo, "A Unified Architecture for A Public Key Cryptographic Coprocessor" submitted to *Journal of System and Architecture*, Second revision, 2007.
2. [Wang07b] Yi Wang, Douglas L. Maskell, Jussipekka Leiwo and Thambipillai Srikanthan, "Efficient DPA-resistant High Radix Modular Exponentiation" submitted to *IEEE Transactions on Circuits and System I*, Second revision, 2007.
3. [Wang06a] Yi Wang, Douglas L. Maskell, Jussipekka Leiwo and Thambipillai Srikanthan, "Unified Signed-Digit Number Adder for RSA and ECC Public-key Cryptosystems", *IEEE Asia Pacific Conference on Circuits and System (APCCAS)*, pp. 1655 – 1658, Dec., 2006.
4. [Wang06b] Yi Wang, Jussipekka Leiwo, Thambipillai Srikanthan and Luo Jianwen, "An Efficient Algorithm for DPA-resistant RSA", *IEEE Asia Pacific Conference on Circuits and System, APCCAS*, pp. 1659-1662, Dec., 2006.
5. [Wang06c] Yi Wang, Jussipekka Leiwo, Thambipillai Srikanthan and Yu Yu, "FPGA based DPA-resistant Unified Architecture for Signcryption", *Third International Conference on Information Technology: New Generations, ITNG 2006*, pp. 571 – 572, Apr., 2006.

6. [Wang05a] Yi Wang, Jussipekka Leiwo and Thambipillai Srikanthan, “A unified architecture for crypto-processing in embedded systems”, *Second International Conference on Embedded Software and Systems*, pp. 37 - 43, Dec., 2005.
7. [Wang05b] Yi Wang, Jussipekka Leiwo and Thambipillai Srikanthan, “Efficient High Radix Modular Multiplication for High-speed Computing in Re-configurable Hardware”, *IEEE International Symposium on Circuits and Systems (ISCAS)*, pp., 1226 – 1229, May, 2005.

1.4 Thesis Organization

This thesis is arranged into seven chapters. In chapter 1, we present the motivation for our research work, the objective, followed by the major contributions of this thesis.

In chapter 2, we provide a literature review on public key cryptosystems. RSA and ECC cryptosystems are introduced. Signcryption schemes for the efficient realization of signature-encryption schemes are also described. We review existing cryptographic coprocessor techniques on various methods for implementing high performance public key cryptographic computations. We also analyze and compare the existing methods to point out the importance of achieving better performance. A relatively new form of cryptanalysis, an attack aimed at the hardware implementation and termed a side channel attack, is reviewed. Various techniques including SPA, DPA, ZPA and RPA are discussed. The existing countermeasures to these attacks are reviewed and discussed.

In chapter 3, we propose a modified Montgomery's modular multiplication algorithm for different key lengths and radix values. We firstly introduce the Montgomery modular multiplication algorithm and other algorithms based on it. We then propose a modified Montgomery modular multiplication algorithm. A pipelined methodology is applied to the proposed architecture. The redundant representations of the partial products generated during the computation are used to reduce the computation time, with the redundant result being converted back into a normal representation in the last step. We also compare our results with a number of other implementations which show that our proposed modular multiplication architecture has better performance than these existing solutions.

In chapter 4, we propose a unified public key cryptographic arithmetic unit that is able to calculate all the necessary operations to support RSA and ECC on $GF(p)$ and $GF(2^p)$ fields. The proposed arithmetic unit focuses on high speed and low cost modular multiplication, as this is the critical operation for RSA and ECC. A dual-field

arithmetic unit is proposed to meet the above requirements including a dual-field addition/subtraction unit, a dual-field modular multiplication unit and a dual-field inversion unit. A new dual-field SD adder is proposed for carry-propagation and carry-less modes of operation using the SD number representation. The proposed adder only needs a two level LUT to produce a carry, resulting in a significant shortening of the critical path and a corresponding improvement in the performance.

In chapter 5, we introduce a power analysis attack which is an efficient way to break general RSA and ECC encryption implementations. A new power analysis attack, called an “initial value DPA” attack, is proposed to break Kim's countermeasure. We also propose a novel countermeasure for “initial value DPA”, DPA and SPA attacks for ECC and show how this countermeasure can be extended to RSA. Finally, dynamic power analysis of the proposed algorithm demonstrates that it is superior to Kim's technique for minimizing power variations, thus making it less susceptible to power attacks

In chapter 6, we develop a framework for a unified cryptographic coprocessor for signing, encryption and signcryption with high performance and area-efficiency. The proposed coprocessor can resist probing attacks of the internal bus and memory. Private keys are maintained within the execution unit to reduce the possibility of leakage during cryptographic operations. The advantage of the proposed architecture is the unification of the operations over $GF(p)$ and $GF(2^p)$, and its ability to resist side channel attacks.

In chapter 7, we present our conclusions, identify areas for potential future work and make recommendations for further research in this area.

Chapter 2 Background and Literature Review

The growing demands for information security are characterizing many communication and computer systems. In order to secure a system, various cryptographic algorithms and protocols are required. These algorithms and protocols can be implemented in hardware or software. However, hardware appears to be the ultimate choice when considering security, performance and efficiency. Hence, more and more cryptographic algorithms and protocols are implemented into hardware. These specialized hardware modules or coprocessors, referred to as cryptographic coprocessors or crypto-processors, have presented new and different opportunities for cryptanalysis attacks. Rather than trying to break the cryptographic algorithm, attackers are focusing on attacking deficiencies in the hardware implementation. These attacks, called side channel attacks, are a powerful method to recover the sensitive data of smart cards or physical devices from timing, power and fault analysis etc. Hence, the ability to defend against cryptanalysis of the hardware must also be considered.

In the following sections, public key cryptosystems, and in particular the two most common cryptosystems, RSA and ECC, are discussed. Various definitions and previous research in this area is introduced. Signcryption schemes for the efficient realization of signature-encryption are also described. Existing cryptographic coprocessor techniques are also discussed and compared. Finally, various side channel attack techniques are introduced, especial power analysis attacks such as SPA, DPA, ZPA and RPA.

2.1 Public Key Cryptosystems

In 1976, Diffie and Hellman [Diff76] first introduced the idea of public key cryptography (PKC). The paper described the key concepts of PKC, including the production of digital signatures. They used this concept to eliminate the need for prior agreement of a secret key to exchange confidential information. Digital signatures were introduced which allowed a message to be uniquely bound to its sender. Since then, numerous public-key cryptosystems have been proposed which base their security on the difficulty of a particular mathematical problem. A widely used public key algorithm is RSA, named after the initials of its inventors, Rivest, Shamir, and Adleman [Rive78]. RSA depends on the difficulty of factoring the product of two very large prime numbers. The Elliptic Curve Cryptosystem (ECC) is another public key algorithm which was proposed by Koblitz [Kobl87] and Miller [Mill85]. The elliptic

curve cryptosystem is a viable alternative to RSA due to its shorter key lengths for the same level of security.

2.1.1 RSA Cryptosystems

2.1.1.1 The Definition of the RSA Algorithm

The RSA [Rive78] cryptosystem is a public-key cryptosystem that offers both encryption and digital signatures (authentication). Public-key cryptosystems use two different keys. One is public (the public key) while the other is kept secret (the private key). Clearly, it is required that computing the private key from the public key has to be intractable [Piep03]. RSA, like any cryptosystem, consists of set-up (parameter generation) and signature (or other cryptogram) computations [Stin02]. The encryption and decryption algorithms using RSA including key generation for RSA are shown in Algorithm 2.1 and Algorithm 2.2 [Mene97].

Algorithm 2.1 Key generation for RSA public-key encryption [Mene97]

Summary: Each entity creates an RSA public key and a corresponding private key. Each entity A should do the following:

-
1. Generate two large random (and distinct) primes p and q , each roughly the same size.
 2. Compute $M=pq$ and $\phi = (p-1)(q-1)$.
 3. Select a random integer b , $1 < b < \phi$, such that $\gcd(b, \phi) = 1$.
 4. Use the extended Euclidean algorithm to compute the unique integer e , $1 < e < \phi$, such that $be \equiv 1 \pmod{\phi}$.
 5. A 's public key is (M, b) ; A 's private key is e .
-

Algorithm 2.2 RSA public-key encryption [Mene97]

Summary: Party B encrypts a message (plaintext) C for party A , which party A decrypts.

-
1. Encryption. B should do the following:
 - (a) Obtain A 's authentic public key (M, b) .
 - (b) Represent the message (plaintext) as an integer C in the interval $[0, M-1]$.
 - (c) Compute $d = C^b \bmod M$.
 - (d) Send the ciphertext d to A .
 2. Decryption. To recover plaintext C from d , A should do the following:
 - (a) Use the private key e to recover $C = d^e \bmod M$.
-

Message transformation between parties A and B involves using these RSA keys. B receives the authentic public key (M, b) from A and computes $d = C^b \bmod M$, where $C \in \{0, M-1\}$ is the message to be signed or encrypted. Then B sends d to A . A computes $C = d^e \bmod M$ using private key e to recover the message C . RSA is mainly concerned with modular exponentiations in the prime field. The speed of modular multiplication is critical to the performance of modular exponentiation.

2.1.1.2 Modular Exponentiation

The main cost operation in RSA is modular exponentiation. A basic algorithm for modular exponentiation is presented in Algorithm 2.3, and consists of modular squaring (multiplication) and modular multiplication. In 1985, Montgomery [Mont85] proposed an attractive method for a fast hardware implementation of the modular multiplication algorithm. Algorithm 2.4 shows the procedure for Montgomery modular multiplication [Mene97]. This method is very efficient and is the basis of many implementations of modular multiplication in hardware as well as software.

Algorithm 2.3 Basic modular exponentiation

Input: d, e, M
Output: $C = d^e \bmod M$

```

1. For  $i = 0$  To  $n-1$  Do
     $C = d \cdot d \bmod M$ 
2.   If  $e_i = 1$  Then
        $C = d \cdot C \bmod M$ 
   EndIf
EndFor

```

Algorithm 2.4 Montgomery modular multiplication [Mene97]

Input: $M = m_{n-1} \dots m_1 m_0$, $m_i \in \{0,1\}$, $A = a_{n-1} \dots a_1 a_0$, $a_i \in \{0,1\}$,
 $B = b_{n-1} \dots b_1 b_0$, $b_i \in \{0,1\}$, $0 \leq A, B < M$, $M < R = 2^n$,
 $\gcd(M, 2) = 1$, $M' = -M^{-1} \bmod 2$
Output: $A \cdot B \cdot R^{-1} \bmod M$

```

1.  $S_0 \leftarrow 0$ .
2. For  $i$  From  $0$  to  $n-1$  Do
     $q_i = (S_i + a_i B) M' \bmod 2$ 
     $S_{i+1} = (S_i + a_i B + q_i m) / 2$ 
3. If  $S_{n-1} \geq M$  Then  $S_{n-1} \leftarrow S_{n-1} - M$ 
4. Return  $S_{n-1}$ .

```

2.1.1.3 The State of Art for RSA Hardware Implementation

Many architectures for implementing modular exponentiation are based on Montgomery's algorithm. The systolic array is the most common structure, with the differences between implementations essentially being the different approaches for avoiding long carry chains. The systolic array, initially proposed by Atrubin [Atru65], is one of the best solutions for modular multiplication with long integers. A simple example is given in Figure 2.1, where two positive integers, to be multiplied, are clocked serially into the first PE of the array, least significant bit first. The product is produced serially by the first PE, least significant bit first, in the same clock cycle.

Figure 2.1. Systolic array with processing elements (PE) [Atru65]

Early systolic array hardware implementations for modular multiplication are proposed in [Even91, Walt93, Eldr93, Korn94] and [Iwam94]. Even [Even91], used Atrubin's multiplier [Atru65] to realize a systolic array for achieving modular reduction based on Montgomery's multiplication algorithm. The systolic array presented by Walter [Walt93] has a throughput of one modular multiplication every clock cycle with a latency of $2n + 2$ cycles for n -digit multiplicands. Eldridge and Walter [Eldr93] improved Montgomery's algorithm by utilizing a novel architecture and achieved a speed almost twice as fast as previous architectures. A linear, purely systolic array forming a digit-serial multiplier for unsigned or 2's complement operands was proposed by Kornerup [Korn94]. Only $n/2$ cells are needed for a full n by n multiply because each cell produces two "digit-produce" terms and accumulates these into a sum S of the same weight. Two types of systolic arrays that implement Montgomery's algorithm without conditional subtraction of the modulus are proposed by Iwamura *et al.* [Iwam94].

Some recent work implementing modular exponentiation on FPGAs is proposed by Blum and Paar [Blum99, Blum01]. They gave absolute area and timing measures for their architectures with different key lengths when implemented on commercially available FPGAs. A full modular exponentiation is computed in $2(n+2)(m+10)$ clock cycles with an n -bit exponent and an m -digit modulus [Blum01]. However, their method lacks generalization as it is only suitable for a limited series of FPGAs.

The Residue Number System (RNS) based on the Chinese Remainder Theorem (CRT) provides a good solution for very long integer arithmetic [Alia91, Posc95, Baja98 Noza01] and [Cho01]. Alia and Martinelli [Alia91] proposed a VLSI implementation using RNS. Their proposed modulo m multiplier was more efficient than ROM-based designs with very long computation parameters. A combination of RNS with efficient modulo reduction methods was proposed by Posch and Posch [Posc95]. Two different implementations of RNS based Montgomery modular multiplication were proposed in [Baja98], one based on processors attached to a broadcast bus, and the other on an oriented ring structure. Nozaki *et al.* [Noza01] developed an RSA hardware implementation based on the RNS implementation of Montgomery's algorithm which discussed how to choose the base size of RNS and the number of parallel processing elements. Cho *et al.* [Cho01] discussed a new radix-4 modular multiplication algorithm based on a sign estimation technique which shortened the computation time for RSA cryptosystems.

Some efficient methodologies have been proposed for the hardware implementation of modular exponentiation and modular multiplication. Montgomery [Mont85] proposed a fast algorithm for modular exponentiation. Eldridge and Walter [Eldr93] improved Montgomery's algorithm by utilizing a novel architecture which achieved almost a two times speed improvement. Yang *et al.* [Yang98] achieved high speed and performance by using a shorter critical datapath. Su *et al.* [Su99] proposed a modification which needs only one addition per iteration instead of the usual two additions, thus the computational time is half that of the previous methods. For very large key lengths, Nozaki [Noza01] provided an RSA hardware implementation based on the RNS Montgomery algorithm which discussed how to choose the base size of RNS and the number of parallel processing elements. The Chinese Remainder Theorem (CRT) was also used to accelerate the speed [Noza01]. Cho *et al.* [Cho01] discussed a new radix-4 modular multiplication algorithm based on a sign estimation technique which shortened the computation time for an RSA cryptosystem. Daly and Marnane [Daly02] reviewed existing architectures for the implementation of Montgomery modular multiplication and exponentiation on FPGA and described how the underlying architecture of the target FPGA device might be utilized to produce an optimum design. Batina and Muurling [Bati02] proposed a scalable systolic array architecture for implementing modular multiplication and modular exponentiation. The speed for computing large key lengths up to 2048 bits can be increased, but at the expense of a significant increase in hardware resources. In [Nibo03], Nibouche *et al.* proposed a scalable architecture for modular multiplication which supported different key lengths and parallelism. Tenca and Koc [Tenc03] proposed a scalable architecture for modular

multiplication which achieved a higher performance within the limitations of the hardware area. A novel hardware architecture, the Coarsely Integrated Hybrid Scanning (CIHS) algorithm, for Montgomery modular multiplication was described in [Mclo04]. The disadvantage of this method was the smaller key length operations which are not suitable for security purposes. Bajard and Imbert [Baja04] proposed an efficient way to implement RSA using the residue number system based on two different bases. Unfortunately, it required significant computational effort to split the number according to the bases and to convert the partial product back into the final result. Kaihara and Takagi [Kaih05] proposed a mixed 4/2 algorithm for modular multiplication/division that required less hardware than that necessary to implement both multiplier and divider separately, but is limited to radix values of 4 or 2.

2.1.2 Elliptic Curve Cryptosystems (ECC)

An elliptic curve cryptosystem (ECC) is another type of public cryptosystem whose solution is related to the solution of the discrete logarithm problem. A finite field or Galois field is an algebraic structure with a finite set of elements. GF_p represents two types of Galois fields: fields of odd characteristic $GF(p)$, where $p > 3$ is a large prime number) and fields of characteristic two $GF(2^p)$, where $q = p$ or $q = 2^p$.

The elliptic curve is defined by constants a and b used in its defining equation. Finally, the cyclic subgroup is defined by its generator (base point) G [Mene97]. For a cryptographic application, the order of G , called $\#E(GF(q))$, that is the smallest non-negative number n such that $nG = O$, must be prime [Mene97]. Suppose that the base point G on a curve E has the prime order r , where r^2 does not divide the order of the curve $\#E(GF(q))$, a key pair can be defined as:

- (1) The private key s is an integer modulo r .
- (2) The corresponding public key W is a point over E defined by $W=sG$.

Party B computes $c=P_m + kW$, using public key W , where $P_m \in E$ is the message, k is a random positive integer, and c is the encrypted text (ciphertext). Party A computes $P_m = c - skG$ using private key s to recover the message. The above ECC encryption involves point addition and point doubling [Piep03]. Point addition takes two points on the curve and constructs another. If these two points are the same point, the computation becomes point doubling. The operations of ECC point addition and point doubling need modular addition/subtraction, modular multiplication and modular inversion over $GF(p)$ and $GF(2^p)$. The formulas for point addition and point doubling, and the fields that they operate on are defined in the following subsections.

2.1.2.1 A Curve Defined over $GF(p)$

A curve defined over a Galois field $GF(p)$, called $E(GF(p))$ ($p > 3$), is a collection of points $(x, y) \in GF(p)$ satisfying a simplified Weierstraß equation of the form [Piep03]:

$$y^2 = x^3 + ax + b \quad (2.1)$$

where $4a^3 + 27b^2 \neq 0$, $a \in GF(p)$ and $b \in GF(p)$.

Let $P = (x_1, y_1)$, $Q = (x_2, y_2)$ and $P + Q = (x_3, y_3)$ be the points over $E(GF(p))$.

Then the point addition and point doubling operations over $GF(p)$ are defined as:

Addition Formula (ECADD(P, Q))

$$x_3 = \lambda^2 - x_1 - x_2 \quad (2.2)$$

$$y_3 = \lambda(x_1 - x_3) - y_1 \quad (2.3)$$

where $\lambda = \frac{y_2 - y_1}{x_2 - x_1}$.

Doubling Formula (ECDBL(P))

$$x_3 = \lambda^2 - 2x_1 \quad (2.4)$$

$$y_3 = \lambda(x_1 - x_3) - y_1 \quad (2.5)$$

where $\lambda = \frac{2x_1 + a}{2y_1}$

2.1.2.2 A Curve Defined over $GF(2^p)$

In Galois field $GF(2^p)$, the definition of $E(GF(2^p))$ can be written as [Piep03]:

$$y^2 + xy = x^3 + ax^2 + b \quad (2.6)$$

Where $a \in GF(2^p)$ and $b \in GF(2^p)$ are constants and $b \neq 0$.

Let $P = (x_1, y_1)$, $Q = (x_2, y_2)$ and $P + Q = (x_3, y_3)$ be the points over $E(GF(2^p))$.

Then the point addition and point doubling operations over $GF(2^p)$ are defined as:

Addition Formula (ECADD(P, Q))

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a \quad (2.7)$$

$$y_3 = \lambda(x_1 + x_3) + y_1 \quad (2.8)$$

where $\lambda = \frac{y_2 + y_1}{x_2 + x_1}$.

Doubling Formula (ECDBL(P))

$$x_3 = \lambda^2 + \lambda + a \quad (2.9)$$

$$y_3 = x_1^2 + (\lambda + 1)x_3 \quad (2.10)$$

where $\lambda = x_1 + \frac{y_1}{x_1}$

The operations of ECC point adding and doubling need modular addition/subtraction, modular multiplication and modular inversion on $GF(p)$ and $GF(2^p)$. The performance of ECC relies predominately on the speed of modular multiplication. An elliptic curve may be represented in other types of coordinates, for example, the project coordinate is commonly used for representing an elliptic curve. The main advantage of point addition in the project coordinate is that it uses only field multiplications, with no inversions required. Thus, inversion only needs to be performed at the end of a point multiplication. More details about different coordinate types representing an elliptic curve can be found in Appendix A.

2.1.2.3 The State of the Art for ECC Hardware Implementation

Modular multiplication in a finite field is similar to that of an RSA implementation. The only difference is that a shorter key size is used. Modular inversion is often performed by the Extended Euclidean Algorithm (EEA) [Mene97]. Modular addition and subtraction in $GF(2^p)$ are performed by bit-wise XORing the coefficients of polynomials $a(x)$ and $b(x)$, where

$$a(x) = \sum_{i=0}^{n-1} a_i x^i, \quad a_i \in GF(2) \quad (2.11)$$

$$b(x) = \sum_{i=0}^{n-1} b_i x^i, \quad b_i \in GF(2) \quad (2.12)$$

Montgomery multiplication has been extended to finite field $GF(2^p)$ in [Koc98], and is achieved by multiplying the corresponding polynomials $(a(x), b(x))$ modulo irreducible polynomial $m(x)$. The inversion of an element $a(x) \in GF(2^p)$ can also be calculated by the Extended Euclidean Algorithm for polynomials.

Several designs are proposed for modular multiplication of ECC. In [Mene97], modular multiplication was extended to compute elliptic scalar multiplications over a finite field. These algorithms were based on the use of precomputations and methods for recoding the multiplier. In [Gord98], several elliptic curve algorithms were analyzed under various conditions. Lopez and Dahab [Lope99] proposed a method to speed up the computation without using precomputations based on Montgomery's method [Mont87]. Gao *et al.* [Gao99] proposed a compact fast elliptic curve scalar multiplier with variable key size. This design reduced the overhead of data transfer between the hardware accelerator and the host and thus achieved high performance. The disadvantage was that the speed of computation remained almost the same when compared to other techniques. Moon *et al.* [Moon01] proposed a new method for computing a fast hardware based arithmetic algorithm for secure data encryption with

a computation time nearly half that of the other methods. In [Nguy03], Nguyen *et al.* proposed a reconfigurable architecture for elliptic curve calculations. Satoh and Takano [Sato03] developed an elliptic curve cryptographic processor supporting Galois fields $GF(p)$ and $GF(2^p)$ which achieved higher speed than the other methods.

2.1.3 Discrete Logarithm Based Cryptosystems

2.1.3.1 ElGamal Encryption and the ElGamal Signature Scheme

Many public key cryptosystems are based on discrete logarithm problems. These cryptosystems include Diffie-Hellman key agreement [Diff76], ElGamal encryption and the ElGamal signature scheme [ElGa85a] and [ElGa85b]. The basic version of these protocols are given in Protocol 2.1 and Protocol 2.2:

Protocol 2.1: Diffie-Hellman key agreement (basic version)

SUMMARY: A and B each send the other a message over an open channel.

RESULT: The shared secret K is known to both parties A and B .

- 1) One-time setup. An appropriate prime p and generator α of Z_p ($2 \leq \alpha \leq p-2$) are selected and published.
 - 2) Protocol message.

$A \rightarrow B : \alpha^x \bmod p$
(1)

$A \leftarrow B : \alpha^y \bmod p$
(2)
 - 3) Protocol actions. Perform the following steps whenever a shared key is required.
 - a) A chooses a random secret x , $1 \leq x \leq p-2$, and sends B message (1).
 - b) B chooses a random secret y , $1 \leq y \leq p-2$, and sends A message (2).
 - c) B receives α^x and computes the shared key as $K = (\alpha^x)^y \bmod p$.
 - d) A receives α^y and computes the shared key as $K = (\alpha^y)^x \bmod p$.
-

Protocol 2.2: ElGamal public-key encryption scheme

SUMMARY: B encrypts a message m for A , which A decrypts.

1. Encryption. B should do the following:
 - a) Obtain A 's authentic public key (p, α, α^a) .
 - b) Represent the message as an integer m in the range $\{0, 1, \dots, p-1\}$.
 - c) Select a random integer k , $1 \leq k \leq p-2$.
 - d) Compute $\gamma = \alpha^k \bmod p$ and $\delta = m \cdot (\alpha^a)^k \bmod p$.
 - e) Send the ciphertext $c = (\gamma, \delta)$ to A .
 2. Decryption. To recover the plaintext m from c , A should do the following:
 - a) Use the private key a to compute $\gamma^{p-1-a} \bmod p$ (note: $\gamma^{p-1-a} = \gamma^{-a} = \alpha^{-ak}$).
 - b) Recover m by computing $\gamma^{-a} \cdot \delta \bmod p$.
-

2.1.3.2 Signcryption

Signcryption is a cryptographic primitive combined digital signing and encryption [Zhen97]. Protocol 2.3 shows the encryption scheme, Protocol 2.4 shows the signature scheme and Protocol 2.5 shows the signcryption scheme which combines the encryption scheme and the signature scheme together.

Protocol 2.3: Encryption scheme [An02]

Definition 2.1 An *encryption scheme* E consists of three algorithms $E = (EGen, Enc, Dec)$.

-
- (a) $EGen(1^k)$, where k is the security parameter, outputs a pair of keys (EK, DK) . EK is the encryption key, which is made public, and DK is the decryption key which is kept secret.
 - (b) The randomized encryption algorithm Enc takes as input a key EK and a message m from the associated message space M and outputs a ciphertext e , defined as $e = Enc_{EK}(m)$.
 - (c) The deterministic decryption algorithm Dec takes as input the ciphertext e , the secret key DK and outputs some message $m \in M$, defined as $m = Dec_{DK}(e)$. We require that $Dec(Enc(m)) = m$, for any $m \in M$.
-

Protocol 2.4: Signature scheme [An02]

Definition 2.2 A *signature scheme* S consists of three algorithms $S = (SGen, Sig, Ver)$.

-
- (a) $SGen(1^k)$, where k is the security parameter, outputs a pair of keys (SK, VK) . SK is the signing key, which is kept secret and VK is the verification key which is made public.
 - (b) The randomized signing algorithm Sig takes as input a secret key SK and a message m from the associated message space M and outputs a signature s , defined as $s = Sig_{SK}(m)$. We will assume that the message m can be determined from the signature s , defined as $m = Msg(s)$.
 - (c) The deterministic decryption algorithm Ver takes as input the signature s , the public key VK and outputs an answer a which is either **succeed** (the signature is valid) or **fail** (the signature is invalid), defined as $a = Ver_{VK}(s)$. We require that $Ver(Sig(m)) = \text{succeed}$, for any $m \in M$.
-

Protocol 2.5: Signcryption scheme [An02]

Definition 2.3 A *signcryption scheme* SE consists of three algorithms. $SE = (SEGen, SigEnc, VerDec)$.

-
- (a) The algorithm $SEGen(1^k)$, where k is the security parameter, outputs a pair of keys (SDK, VEK) . SDK is the user's sign/decrypt key, which is kept secret, and VEK the user's verify/encrypt key, which is made public. Note, that in the signcryption setting,

- all participating parties need to invoke *Gen*. For a user P , denote P 's keys by SDK_P and VEK_P .
- (b) The randomized signcryption (sign/encrypt) algorithm *SigEnc* takes as input the secret key SDK_s of sender s , the public key VEK_p of receiver r and a message m from the associated message space M . This operation is denoted by $u = \text{SigDnc}_{SDK_s, VEK_r}(m)$, or simply $u = \text{SigDnc}(m)$ if it is clear which keys are being used.
 - (c) The deterministic de-signcryption (verify/decrypt) algorithm *VerDec* takes as input the signcryption (ciphertext) e , the secret key VEK_r of receiver R and the public key VEK_s of sender S and outputs $m \in M \cup \{\perp\}$, where $\perp$ indicates that the message was not encrypted or signed properly. This operation is denoted by $m = \text{VerDec}_{VEK_s, SDK_r}(u)$, or simply $m = \text{VerDec}_r(u)$ if it is clear which keys are being used.

The usual way to provide both confidentiality and authentication is to sequentially compose encryption and signature schemes as "encryption-then-sign" and "sign-then-encryption". A discussion of the efficiency of signcryption can be found in [Zhen97].

2.2 Cryptographic Coprocessors

A higher demand and greater awareness of security problems has resulted in more emphasis being placed on secure, high performance, reliable and flexible systems. A cryptographic coprocessor is a hardware module that includes a processor specialized for encryption and related processing, which supports basic cryptographic algorithms, protocols and schemes.

2.2.1 General Commercial Crypto-coprocessors

A number of commercial cryptographic products are currently available. The Infineon Security chip card SLE88CFX4000P [SCIC04] is a high end 32-bit security controller with 400 Kbytes of EEPROM, 80 Kbytes of ROM, 16 Kbytes of RAM. The coprocessor is ideal for virtual machine execution and for the calculation of cryptographic algorithms, and is optimized for multi-application cards with the highest level of security. Figure 2.2 gives the simplified block diagram of the SLE88CFX4000P coprocessor.

The features of the SLE88CFX4000P include hardware memory management, various sensors for detecting the outside environment, a security optimized layout and hardware encryption of memories, etc. The coprocessor also includes a concept called TrustZone which is concerned with security integrated hardware for resisting software attacks, originally proposed by Alves and Felton of ARM [Alve04].

Figure 2.2. The block diagram of SLE88CFX4000P, 32-bit CPU and peripherals [SCIC04]

Figure 2.3. Providing security against malicious software attack TrustZone [Alve04]

Figure 2.4. The components of an embedded system-on-chip architecture demarcated into secure and non-secure areas [Alve04]

Figure 2.3 shows the concept of the ARM TrustZone structure, which will behave differently for normal applications and for security applications. Figure 2.4 shows the structure of an embedded SOC architecture demarcated into secure and non-secure areas. The black blocks are not used for normal applications unless the applications are sensitive. The advantages of TrustZone are primarily good performance and flexibility. It provides a safe environment for secure data on chip. This enables a complete approach to security. And full bus-bandwidth access is provided to all storage areas to provide fast memory access speeds. In addition, safe local cache data is stored securely in decrypted form providing even faster access. The encrypted data can access the same FLASH memory as the non-secure world, ensuring cheap, large and flexible storage is utilized. It provides flexibility to allow customization and upgrades to the secure system even after the SOC is finalized.

Implementing a tamper-resistance system involves the use of a separate secure co-processor module, which is dedicated to processing all sensitive information in the system. Any sensitive information that needs to be sent out of the secure co-processor is encrypted [IBM02]. Figure 2.5 gives the outline of the IBM 4758 cryptographic coprocessor schematic. The main features of the IBM 4758 PCI cryptographic coprocessor include:

1. A tamper-resistant design certified under FIPS PUB 140-1, suitable for high-security processing and cryptographic operations. The 4758 provides a secure platform on which developers can build secure applications
2. Hardware to perform DES, T-DES, SHA-1, random number generation and modular math functions for RSA and similar public key cryptographic algorithms.
3. Secure code loading that enables updating of the software with the coprocessor installed in the application system.
4. The IBM common cryptographic architecture and public key cryptography standard PKCS#11 implementations as well as custom software options.
5. OEM and end user purchase options.

The above commercial cryptographic coprocessors have the same features for defending against attacks from off-chip and support similar algorithms and protocols.

Figure 2.5. The IBM 4758 cryptographic coprocessor schematic [IBM02]

2.2.2 RSA based Cryptographic Coprocessors

The earliest RSA chip was designed by Rivest [Rive86], and contained 512-bit registers for storage of intermediate results, carry-save adder logic, and shifter logic. Another early RSA processor was designed by Kochanski [Koch85] with a serial/parallel architecture for performing multiplication and modulo reduction. The modular exponentiation operation was performed using a fast integer arithmetic chip and the host microprocessor. A 512-bit modular exponentiation for RSA applications was proposed by Rankine [Rank87], providing a complete single RSA chip supporting a wide range of applications such as key management, smartcard and stream encryption, etc. Vandemeulebroecke *et al.* [Vand90] proposed a 1024-bit RSA cryptographic chip with a new carry-free division algorithm based on Redundant Signed Digit (RSD) arithmetic to avoid the carry propagation.

Recently, there have been many proposals and designs for RSA public key cryptosystems. The key kernel to realize faster RSA computation still lies in the performance of modular exponentiation. In [Royo97], Royo *et al.* proposed a coprocessor for cryptography applications like the Digital Signature Algorithm (DSA). The coprocessor used a special set of instructions for dealing with high accuracy integers, as well as for the rapid evaluation of modular multiplications and exponentiations. The methodology and design of a scalable Montgomery multiplication module is introduced by Tencu and Koc [Tenc99, Tenc01] and [Tenc03]. A word-based radix-2 algorithm is proposed in [Tenc99] to demonstrate the scalability of pipelined stages and word sizes. In [Tenc01], Tencu *et al.* proposed a pipelined Montgomery multiplier which has the ability to work on any given operand precision and adjust to the available area and performance. The multiplier in [Tenc03] was able to work with any precision of the input operands, limited only by memory or control constraints and the proposed architecture had enough freedom to select the word size and the degree of parallelism to be used according to the available area and/or desired performance. More results with radices ranging from 1 to 32 and word sizes ranging from 4 to 128 were reported. Goodman and Chandrakasan [Good01] proposed a public key cryptography processor using a modular exponentiation algorithm on reconfigurable hardware, which focused on an energy efficient reconfigurable hardware implementation. Kitsos *et al.* [Kits02] proposed a method supporting digital signature including a hash function and RSA algorithms. Kim and Lee [Kim04a] proposed an architecture to support cryptographic algorithms which included a 32-bit RISC processor block and a coprocessor block consisting of AES, Kasumi, SEED, Triple-DES private key cryptographic algorithms, EC and RSA public key cryptographic algorithms. The coprocessor allowed the execution of encryption, decryption and key scheduling operations. The 32-bit RISC processor block was used to execute non trusted algorithms such as hash functions etc. The structure of the Design is shown in Figure 2.6. The disadvantage of the architecture proposed by Kim and Lee was the large implementation area and the fixed bit width.

2.2.3 ECC based Cryptographic Coprocessors

The first reported results for performing elliptic curve operations on dedicated hardware was by Agnew *et al.* [Agne90]. The proposed architecture included a normal basis multiplier as the arithmetic unit and a Motorola M68008 processor as the control unit. Later, a VLSI implementation of an arithmetic processor in $GF(2^{155})$ is discussed to efficiently support Diffie-Hellman and ElGamal protocols [Suti98a].

Figure 2.6. The block diagram of a private and public key crypto-processor [Kim04a]

Recently, a number of ECC coprocessors have been proposed [Suti98a, Suti98b, Gao99, Okad00, Orla00, Orla01, Moon01, Bati02, Good01] and [Erns03]. Sutikno *et al.* [Suti98a] proposed an arithmetic processor over $GF(2^{155})$ for elliptic curve calculation. Their arithmetic processor has a 32-bit wide internal bus and 15 5-bit instructions for the operation codes (The 3 least significant bits (LSB) are used for selecting instructions and the 2 most significant bits (MSB) are for selecting the active register). A VLSI implementation of an elliptic curve cryptosystem for the ElGamal encryption scheme was also proposed by Sutikno *et al.* [Suti98b]. Figure 2.7 shows the structure of Sutikno's arithmetic processor. It consisted of instruction latch, main controller, selector and arithmetic module supporting 32-bit operation. The processor is able to perform an encryption at the rate of 6.5×10^{-4} bits/clock cycle, and a decryption at the rate of 13.1×10^{-4} bits/clock cycle.

Figure 2.7. Architecture for Sutikno's arithmetic processor [Suti98b]

Gao *et al.* [Gao99] proposed an elliptic curve coprocessor with a pipelined digital-serial modified Massey-Omura multiplier for massively parallel computation. Okada and Orlando [Okad00] proposed a scalable bit parallel structure for minimizing the computation time. A number of improvements for ECC operations on $GF(p)$ were proposed in [Okad00, Moon01] and [Bati02] including modular addition, modular subtraction, modular multiplication and modular inversion. Low power designs for an ECC coprocessor were proposed in [Good01] and [Erns03]. Goodman [Good01] proposed an energy-efficient reconfigurable public key cryptography processor for key sizes ranging from 0 to 1024. Ernst [Erns03] used a HW/SW system to save power consumption. Unfortunately, these designs both lacked flexibility.

2.2.4 A Unified Architecture for Cryptographic Coprocessor

Some considerable research effort has already been expended on the efficient unification of modular multiplication, modular addition/subtraction and modular inversion over both $GF(p)$ and $GF(2^p)$ [Sava00, GroB01, Wolk02, Sato03, Gutu03, Bati04, Cila05] and [Crow05]. The difference between modular multiplication over $GF(p)$ and $GF(2^p)$ lies in the addition arithmetic, and in particular, whether the carry of the summand is forwarded or not. A modified full adder supporting carry propagation and carry-less modes of operation is commonly used in existing designs. However, this results in a critical path delay difference between the two modes of operation which affects the performance. For instance, in a conventional logic implementation of a full adder, the critical path has three simple gate delays for the carry and two XOR gate delays to produce the result. This contrasts to a critical path of just a single XOR gate delay for a carry-less mode of operation. This modified full adder approach has been applied in [Sava00, GroB01, Wolk02, Cila05] and [Crow05] to adopt the modular operations over both $GF(p)$ and $GF(2^p)$.

Savas *et al.* [Sava00] used a pipelined methodology to obtain a scalable and unified architecture for a Montgomery multiplication module which operates in both the $GF(p)$ and $GF(2^p)$ finite fields. The structure of a full adder and Savas's dual-field adder is shown in Figure 2.8. A fixed-area multiplication module can handle operands of any size and the word size can be selected based on the area and performance requirement. The dual-field adder is basically a full adder which is capable of performing carry propagation and carry-less operations in both fields. The dual-field adder has a field select enable (FSEL) input which controls the carry propagation within the adder. When FSEL=1, the adder performs the bit-wise addition with carry which enables the multiplier to do arithmetic in the $GF(p)$ field. When FSEL=0, the output C_{out} is forced to 0 regardless of the values of the inputs.

Figure 2.8. The structure of (a) a full adder and (b) a dual-field adder [Sava00]

In [GroB01], a bit-serial multiplier was proposed that resulted in less hardware resources, but with an increased computation time. Batina *et al.* [Bati02] proposed an ECC processor supporting RSA and ECC on the same platform but only over $GF(p)$. The proposed architecture has two large number modular Montgomery units for performing modular multiplication, one large number co-processor for performing modular addition/subtraction and a memory for scheduling operations. A systolic array structure is used for Montgomery modular multiplication. A scalable unified architecture for modular inversion was reported in [Gutu03]. Wolkerstorfer [Wolk02] proposed a dual-field arithmetic unit that offered all instructions to implement elliptic curve digital signature over prime fields $GF(p)$ and binary extension fields. The arithmetic unit supported operations such as shift, increment, comparison, addition, multiplication and inverse using the extended Euclidean algorithm. The basic architecture of Wolkerstorfer's arithmetic unit is shown in Figure 2.9.

Figure 2.9. The architecture for a dual-field arithmetic unit [Wolk02]

Other unified architectures based on the modified carry-save adder have been proposed by Cilaro *et al.* [Cila05] and Crowe *et al.* [Crow05]. Currently, the best performance reported for modular operations over both $GF(p)$ and $GF(2^p)$ was in [Sato03]. This implementation used a parallel dual-field unified Montgomery multiplier and an on-the-fly redundant binary converter. The design used 117.5K gates with a 64-bit multiplier and achieved operation times of 1.21 ms and 0.19 ms, respectively, for a 160-bit EC scalar multiplication over $GF(p)$ and $GF(2^p)$ in a 0.13 μm CMOS technology.

2.3 Side Channel Attacks

Security is a requirement for an increasing number of embedded systems. With the well established trend for developing fast hardware solutions in the form of embedded cryptographic co-processors, there have been a number of techniques proposed to “crack” these systems. While many theoretically secure cryptographic techniques exist, the real-world attacks are usually against the implementation of the techniques and not the theoretical properties of the techniques. Such attacks are called side channel attacks [BarEl]. These attacks that are based on “side channel information”, that is, information that can be retrieved from the encryption device that is neither the plaintext to be encrypted nor the ciphertext resulting from the encryption process. These attacks are aimed at the physical implementation of the cryptographic algorithms. Thus the design of a secure application specific instruction-set processor (ASIP) must have the ability to defend against these kinds of attacks.

Unfortunately, hardware encryption devices exhibit a significant amount of information that is easily measurable. This includes radiation of various sorts, power consumption, etc. Other encryption devices also have additional inputs such as voltage that can be modified to cause predictable outcomes. Side channel attacks make use of some or all of this information, along with other cryptanalytic techniques, to recover the secret information.

2.3.1 Power Analysis

Most modern cryptographic devices are implemented using semiconductor logic gates, which are constructed out of transistors [Koch99]. Simple Power Analysis (SPA) is a technique that involves directly interpreting power consumption measurements during cryptographic operations. The sequence of operations of a certain cryptographic algorithm can be revealed by observing the power consumption during the execution.

Therefore, sensitive information like private keys can be easily revealed by a simple power analysis.

Another approach, called Differential Power Analysis (DPA) [Koch99], is based on statistical analysis in which the attacker can guess the correctness of the keys by comparing the differences between a sample power trace and the correct key power trace. Public key algorithms can be analyzed using DPA by correlating candidate values through an intermediate value with power consumption measurements. Zero Power Analysis (ZPA) [Akis03] and Refined Power Analysis (RPA) [Goub03] have been proposed based on the special features of the elliptic curve where the power consumption is zero at certain points. Although efficient and relatively cheap ways of avoiding DPA attacks on scalar multiplication, such as "random projective coordinates" [Akis03], "random elliptic curve isomorphism" [Akis03] or "random field isomorphism" [Akis03] exist, such a DPA-protection of the scalar multiplication is not sufficient for many elliptic curves. Additionally, for a chosen message scenario, an RPA attack is still possible even if one of the three aforementioned countermeasures is used. For instance, suppose the attacker already knows the highest bits $e_{n-1}, \dots, e_{i+1}$ of the secret multiplier e . By applying the chosen message and the special elliptic curve points, it is possible for the attacker to deduce the next bit e_i of secret multiplier e .

An extension of Goubin's attack is the Zero-value Point Attack (ZPA) [Goub03]. Even if a point had no zero-value coordinate, the auxiliary register might take a zero-value. Therefore, it was possible that attackers could find the specific points where the power consumption was zero during the explicit implementation of the addition formula in a normal elliptic curve computation. Zero-value registers cannot be randomized by general methods. Several points $P = (x, y)$ that caused a zero-value register have been found, for instance: $3x^2 + a = 0$; $5x^4 + 2ax^2 - 4bx^2 + a^2 = 0$; P is a y -coordinate self-collision point; etc.

The doubling attack proposed by Fouque and Valette [Fouq03] is based on the fact that similar intermediate values may be manipulated when working with points P and $2P$. The exceptional procedure attack reveals the secret scalar value of the scalar multiplication computation using the error arising from the exceptional procedures [Izu03]. This kind of attack is efficient against the addition formula proposed by Brier-Joye [Brie02].

There are three types of power analysis attacks against the smartcard implementation of modular exponentiation listed in [Mess99] and [Mess02]. The basic premise of the

Single-Exponent, Multiple-Data (SEMD) attack is that by comparing the power signal of an exponentiation using a known exponent to the power signal using an unknown exponent, the adversary can learn where the two exponents differ, and thus determine the secret exponent. A Multiple-Exponent, Single-Data (MESD) attack is more powerful than the SEMD attack, but requires the assumption that the smartcard will maintain a consistent power consumption when using the exponents chosen by the attackers. A Zero-Exponent, Multiple-Data (ZEMD) attack is similar to the MESD attack, with the assumption that the attacker knows basic knowledge about the exponentiation algorithm being used by the smartcard.

A DPA attack against the modular reduction within a Chinese Remainder Theorem (CRT) implementation of RSA uses hypotheses of the remainder after the modular reduction with one of the primes [Boer03]. The basic assumption of this DPA attack is that the distance of the input data equals the distance of the intermediate data after the modular reduction at least for a subgroup of single measurements.

DES, AES and RSA are sensitive to DPA and SPA, while ECC is sensitive to not only DPA and SPA, but also ZPA and RPA, as shown in Table 2.1. The countermeasures for defending against those power analysis attacks need to be taken into account when implementing a cryptographic algorithm in hardware.

Table 2.1. The relationship between power attacks and the common cryptographic algorithms

	DPA	SPA	ZPA	RPA
DES	X	X		
AES	X	X		
RSA	X	X		
ECC	X	X	X	X

2.3.2 Countermeasures against Power Analysis Attacks

As power analysis attacks are efficient ways to attack ECC and RSA based protocols, it is reasonable to pick ECC as an example to illustrate possible countermeasures against power analysis attacks. These countermeasures can later be extended to RSA.

Algorithm 2.5 shows the binary method to compute $Q = eP$. The attackers are able to recover e when $Q_i = eP_i$ is computed for various P_i for $1 \leq i \leq k$ by using an SPA attack to Algorithm 2.5. There are three countermeasures based on introducing random

numbers into the calculations during the computation of $Q = eP$ [Coro99]. Randomization of the private exponent is the first countermeasure and is shown in Algorithm 2.6 [Coro99]. The second countermeasure involves “blinding the point P ” and is shown in Algorithm 2.7. The last countermeasure involves randomizing the projective coordinates and is shown in Algorithm 2.8. The projective coordinates (X, Y, Z) of a point $P = (x, y)$ on an elliptic curve are given by: $x = X/Z$ and $y = Y/Z$ [Mene93].

Algorithm 2.5 The binary method for computing $Q = eP$

Input: e, P

Output: $Q = eP$

1. $Q = P$
 2. **For** $i = n-2$ **Down To** 0 **Do**
 $Q = 2 \cdot Q$
 3. **If** $e_i = 1$ **Then**
 $Q = Q + P$
 End If
 - End For**
 4. **Return** Q
-

Algorithm 2.6. Coron's first countermeasure: Randomizing the exponent [Coro99]

Input: e, P

Output: $Q = eP$

$\#e$ is the number of point on the cure

Computation

1. Select an n bit random number k .
 2. Compute $e' = e + k \cdot \#e$.
 3. Compute the point $Q = e'P$. The output is $Q = eP$, since $\#eP = O$.
-

Algorithm 2.7. Coron's second countermeasure: Blinding the point P [Coro99]

Input: e, P

Output: $Q = eP$

Pre-computation

Select a secret random point R and compute $S = eR$

Computation

1. Compute $P' = R + P$.
2. Compute the point $Q = eP' \cdot S$. The output is $Q = eP$.

Algorithm 2.8. Coron's third countermeasure: Randomized projective coordinates
[Coro99]

Input: e, P

Output: $Q = eP$

-
1. For a projective coordinate representation of a point $P = (X, Y, Z)$, $P' = (X', Y', Z') = (\lambda X, \lambda Y, \lambda Z)$ where λ is a random number and $\lambda \neq 0$ in the finite field.
 2. Compute $P = eP'$.
 3. The output is $Q = eP$.
-

Möller proposed a method for elliptic curve multiplication using a special signed-digit encoding mechanism to ensure that point doubling and point additions occur in uniform patterns [Moll01]. As the basic computation is based on a binary number, the Binary expansion with a Random Initial Point (BRIP) is an efficient countermeasure against the SPA and DPA variants [Mami04]. Algorithm 2.9 shows the BRIP countermeasure in detail.

Algorithm 2.9. Binary expansion with a random initial point

Input: e, P

Output: $Q = eP$

-
1. Select a secret random point R
 2. $T_0 = R, T_1 = -R, T_2 = P - R$
 3. **For** $i = n-1$ **Down To** 0 **Do**
 $T_0 = 2T_0$
If $e_i = 0$ **Then** $T_0 = T_0 + T_1$
 $T_0 = T_0 + T_2$
 4. **Return** $Q = T_0 + T_2$
-

Kim *et al.* [Kim04b] proposed a secure and practical CRT-based RSA algorithm for resisting side channel attacks, including power analysis, timing and fault attacks. Algorithm 2.10 shows the technique to resist against DPA by adding random numbers to avoid the leakage of the real power consumption.

Algorithm 2.10. Kim's exponentiation algorithm which is immune to DPA, SPA and timing attacks [Kim04b]

Input: d, e, M

Output: $C = d^e \bmod M$

1. Choose a random number r .
 2. $C = d \cdot r \bmod M$
 3. $Temp[0] = r^{-1} \bmod M$ and $Temp[1] = d \cdot r^{-1} \bmod M$
 4. **For** $i = n-1$ **Down To** 0
 $C = C^2 \bmod M$
 $C = C \cdot Temp[e_i] \bmod M$
 5. $C = C \cdot Temp[0] \bmod M$
 6. **Return** C
-

2.4 Conclusion

In this chapter, the definition, calculation and hardware implementation of a number of cryptosystems are presented, and in particular the RSA and ECC public key cryptosystems. Existing cryptographic coprocessor techniques are presented. These included RSA coprocessors, ECC coprocessors and a number of unified coprocessors which supported some RSA and ECC operations. With the widespread use of hardware cryptographic coprocessor, a new security threat, termed side channel attacks, has emerged as an effective mechanism for attacking hardware based cryptographic algorithm implementations. Power analysis is normally used for attacking those algorithms in which, SPA, DPA, ZPA and RPA are discussed. Current countermeasures against the common side channel attacks are also given.

Chapter 3 An Efficient Modular Multiplication Algorithm

3.1 Introduction

In RSA cryptosystems, the major challenge is in computing the modular exponentiation, $C = d^e \bmod M$, as efficiently as possible. The Montgomery modular multiplication algorithm is used extensively to implement modular exponentiation [Mont85]. Many hardware implementations of modular multiplication are proposed in [Even91, Walt93, Eldr93, Korn94, Iwam94, Yang98, Blum99, Blum01, Su99, Nibo03, Tenc03, Mclo04] and [Kaih05]. One of the fastest implementations of modular multiplication on FPGA was reported in [Blum01]. Currently, key sizes of up to 1024 or 2048-bits are used to ensure an appropriate level of security for an application. Thus methodologies to shorten the total computation time of the modular exponentiation are more critical than just speeding up the modular multiplication.

This chapter presents a modified Montgomery's modular multiplication algorithm [Wang05b] for different key lengths and radix values. We firstly introduce the Montgomery modular multiplication algorithm and other algorithms based on it. We then proposed a modified Montgomery modular multiplication algorithm based on consecutive binary digits. A pipelined methodology is applied to the proposed architecture. The redundant representations of the partial products generated during the computation are used to minimize the conversion time. The redundant result needs to be converted back into a normal representation in the last step. We also present the results of a number of implementations which show that our proposed modular multiplication architecture has better performance than existing solutions.

3.2 Montgomery Multiplication

Consider the computation of the modular exponentiation $C = d^e \bmod M$, where d is the ciphertext to be decrypted and the private key is e , with e being represented by a binary number of length $n + 1$ as $\{e_n e_{n-1}, \dots, e_1 e_0\}$, and the length of d is n . Timing attacks exploit the observation of execution time for cryptographic computations, which include instruction execution time variation and performance optimization [Koch99]. The normal modular exponentiation is performed as a modular multiplication calculation when $e_i = 1$, with the hardware doing nothing when $e_i = 0$. Thus it is easy to determine the private key e by performing a simple timing attack. One solution is to add a dummy variable which does not effect the result, but makes the hardware perform a calculation when $e_i = 0$. If we let r be a fixed constant of length n used to

resist timing attacks.

$$r = \underbrace{\{00\ldots 001\}}_n \}_2 \quad (3.1)$$

Algorithm 3.1 describes the method to realize the modular exponentiation, $C = d^e \bmod M$, using a sequence of modular multiplications, which is resistant to a simple timing attack.

Algorithm 3.1 Our proposed modular exponentiation against linear timing attacks

Input: M, d, e, r

Output: $C = d^e \bmod M$

```

1.   $C = 1$ 
2.  For  $i = n$  Downto  $0$ 
2.1     $C = C \cdot C \bmod M$ 
2.2    If  $e_i = 1$ 
2.2.1       $C = C \cdot d \bmod M$ 
        Else
2.2.2       $C = C \cdot r \bmod M$ 
        EndIf
    EndFor
```

A time lag can be added to the exponentiation algorithm to resist a linear timing attack by making the calculation time for the algorithm essentially uniform, independent of the value of e_i .

Modular multiplication can be efficiently implemented using Montgomery's modular multiplication algorithm, which uses two n -bit integers A and B , and generates the output $S = A \cdot B \cdot R^{-1} \bmod M$, where $R = 2^n$ and M is an integer in the range $2^{n-1} < M < 2^n$, such that $\gcd(R, M) = 1$. Montgomery multiplication is described in Algorithm 3.2 below. As can be seen from Algorithm 3.2, Montgomery's modular multiplication needs n iterations with two additions per iteration, and with an additional step at the end to convert the final result S_n into the correct range. Also, Montgomery's algorithm calculates $S = A \cdot B \cdot R^{-1} \bmod M$ and thus two separate steps are needed to calculate a modular multiplication as:

$$A \cdot B \bmod M = (A \cdot B \cdot R^{-1} \bmod M) \cdot R^2 \cdot R^{-1} \bmod M \quad (3.2)$$

That is, in the second step, A is substituted by the product from the first step, while B is R^2 . The radix-2 Montgomery multiplication algorithm is simple and is thus well suited to hardware implementation. For completeness, Montgomery's multiplication

algorithm with radix greater than 2 is given in Algorithm 3.3 [Orup95]. High-radix Montgomery modular multiplication needs at most two k bit multiplications and a k bit addition to compute $q_i = (((S_i + a_i B) \bmod 2^k) M') \bmod 2^k$, and two k bit multiplications and two $(n+k)$ bit additions for computing $S_{i+1} = (S_i + q_i M' + a_i B) / 2^k$.

Algorithm 3.2 The radix-2 Montgomery multiplication algorithm [Eldr93]

Input: $M = \sum_{i=1}^{n-1} 2^i m_i, m_i \in \{0,1\}$,
 $B = \sum_{i=1}^{n-1} 2^i b_i, b_i \in \{0,1\}$,
 $A = \sum_{i=1}^{n-1} 2^i a_i, a_i \in \{0,1\}$,
 where $A, B < M, M < R = 2^n, \gcd(2, M) = 1$
Output: $S = A \cdot B \cdot R^{-1} \bmod M$

```

1.  $S_0 = 0$ 
2. For  $i=0$  To  $n-1$ 
2.1  $q_i = (S + a_i B) \bmod 2$ 
2.2  $S_{i+1} = (S_i + q_i M + a_i B) / 2$ 
    EndFor
3. If  $S_n \geq M$  Else  $S_n = S_n - M$  EndIf
    
```

Algorithm 3.3 High radix Montgomery multiplication algorithm [Orup95]

Input: $M = \sum_{i=1}^{n-1} (2^k)^i m_i, m_i \in \{0,1,\dots,2^k-1\}$,
 $B = \sum_{i=1}^{n-1} (2^k)^i b_i, b_i \in \{0,1,\dots,2^k-1\}$,
 $A = \sum_{i=1}^{n-1} (2^k)^i a_i, a_i \in \{0,1,\dots,2^k-1\}$,
 where $A, B < 2M, 4M < R = 2^{kn}$,
 $M' = -M^{-1} \bmod 2^k, \gcd(2^k, M) = 1$
Output: $S = A \cdot B \cdot R^{-1} \bmod M$

```

1.  $S_0 = 0$ 
2. For  $i=0$  To  $n-1$ 
     $q_i = (((S_i + a_i B) \bmod 2^k) M') \bmod 2^k$ 
     $S_{i+1} = (S_i + q_i M' + a_i B) / 2^k$ 
    EndFor
3. If  $S \geq M$  Else  $S = S - M$  EndIf
    
```

3.3 Blum's Modified Montgomery Modular Multiplications

Blum [Blum01] proposed a modified Montgomery modular multiplication to simplify and speed up the computation procedure. For Algorithm 3.2, if two additional iterations of the loop are performed with $a_{n+1} = 0$ and inputs $A, B < 2M$, the inequality $S_{n+2} < 2M$ is satisfied. Now, S_{n+2} can be used as input B for the next modular multiplication, and B can be shifted up by one position. This results in $a_i B \bmod 2 = 0$ and the operation $q_i = (((S_i + a_i B) \bmod 2^k) M') \bmod 2^k$ is avoided. $(S_i + q_i M' + a_i B) / 2^k$ is replaced by $(S_i + q_i M) / 2 + a_i B$. The cost of this simplification is one more execution of the loop with $a_{n+2} = 0$, resulting in a computational complexity associated with the two additions of the n bit operands for computing S_{i+1} . Algorithm 3.4 shows Blum's modified Montgomery modular multiplication with radix 2.

Algorithm 3.4 Blum's radix-2 Montgomery modular multiplication [Blum01]

Input: $M = \sum_{i=1}^{n-1} 2^i m_i, m_i \in \{0,1\},$
 $B = \sum_{i=1}^{n-1} 2^i b_i, b_i \in \{0,1\},$
 $A = \sum_{i=1}^{n-1} 2^i a_i, a_i \in \{0,1\},$
 $a_{n+1} = 0, a_{n+2} = 0,$
 where $A, B < 2M, M < R = 2^{n+2}, \gcd(2, M) = 1$

Output: $S = A \cdot B \cdot R^{-1} \bmod M$

```

1.  $S_0 = 0$ 
2. For  $i=0$  To  $n-1$ 
     $q_i = S_i \bmod 2$ 
     $S_{i+1} = (S_i + q_i M) / 2 + a_i B$ 
EndFor

```

A modified high-radix Montgomery modular multiplication algorithm is also proposed by Blum [Blum01]. In order to reduce the computation complexity, the conditions are changed to $4M < 2^{kn}$ and $A, B < 2M$. This results in $S_n < 2M$ as needed for further processing. The complexity of quotient q_i can be further reduced by replacing B with $B \cdot 2^k$. As $a_i B \bmod 2^k = 0$, $S_{i+1} = (S_i + q_i M' + a_i B) / 2^k$ is reduced to $S_{i+1} = (S_i + q_i M') / 2^k + a_i B$. Blum's modified high-radix Montgomery modular multiplication algorithm is given as Algorithm 3.5.

Algorithm 3.5 Blum's modified high-radix Montgomery modular multiplication [Blum01]

Input: $M = \sum_{i=1}^{n-1} (2^k)^i m_i, m_i \in \{0,1,\dots,2^k-1\},$
 $B = \sum_{i=1}^{n-1} (2^k)^i b_i, b_i \in \{0,1,\dots,2^k-1\},$
 $A = \sum_{i=1}^{n-1} (2^k)^i a_i, a_i \in \{0,1,\dots,2^k-1\},$
 $M < R = 2^{kn}, M' = -M^{-1} \bmod 2^k, \gcd(2^k, M) = 1$
 where $A, B < 2M, 4M < R = 2^{kn},$
Output: $S = A \cdot B \cdot R^{-1} \bmod M$

1. $S_0 = 0$
 2. **For** $i=0$ **To** $n-1$
 $q_i = S_i \bmod 2^k$
 $S_{i+1} = (S_i + q_i M') / 2^k + a_i B$
EndFor

3.4 An Improved Modular Multiplication Algorithm

We propose an improved algorithm for modular multiplication to reduce the number of iterations and hence the computational complexity in Algorithm 3.5. The modified algorithm uses consecutive binary digits, a_{2i-1}, a_{2i} and a_{2i+1} , in A , and is given in Algorithm 3.6.

Algorithm 3.6 The proposed high-radix Montgomery modular multiplication

Input: $M = \sum_{i=1}^{n-1} (2^k)^i m_i, m_i \in \{0,1,\dots,2^k-1\},$
 $B = \sum_{i=1}^{n-1} (2^k)^i b_i, b_i \in \{0,1,\dots,2^k-1\},$
 $A = \sum_{i=1}^{n-1} (2^k)^i a_i, a_i \in \{0,1,\dots,2^k-1\},$
 $a_{n+1} = a_n = a_{-1} = 0, A, B < 2M, 4M < R = 2^{kn},$
 $M < R = 2^{kn}, M' = -M^{-1} \bmod 2^k, \gcd(2^k, M) = 1$
Output: $S = A \cdot B \cdot R^{-1} \bmod M$

1. $S_0 = 0$
 2. **For** $i=0$ **To** $n/2$ **Do**
 2.1 $q_i = (((S_i + (a_{2i-1} + a_{2i} + a_{2i+1})B) \bmod 2^k) M') \bmod 2^k$
 2.2 $S_{i+1} = (S_i + q_i M' + (a_{2i-1} + a_i - 2a_{2i+1})B) / 2^k$
EndFor
 3. **If** $S_{n/2+1} \geq M$ **Else**
 3.1 $S_{n/2+1} = S_{n/2+1} - M$
EndIf

The proof of correctness of the proposed algorithm is as follows:

$$\begin{aligned}
 A &= \sum_{i=1}^{n-1} (2^k)^i a_i \\
 &= 2^k (2^{n-1} a_{n-1} + 2^{n-2} a_{n-2} + \dots + 2a_1 + a_0) \\
 &= 2^k (4^{\frac{n}{2}} (a_{n-1} + a_n - 2a_{n+1}) + \dots + 4(a_1 + a_2 - 2a_3) + (a_{-1} + a_0 - 2a_1)) \\
 &= 2^k (4^{\frac{n}{2}} (a_{\frac{n}{2}-1} + a_{\frac{n}{2}} - 2a_{\frac{n}{2}+1}) + \dots + 4(a_{2,1-1} + a_{2,1} - 2a_{2,1+1}) + (a_{2,0-1} + a_{2,0} - 2a_{2,0+1})) \quad (3.3)
 \end{aligned}$$

Where $a_{-1} = 0$ and the highest bits a_n and a_{n+1} are equal to 0. Therefore, after $n/2 + 1$ cycles, the result is:

$$2^k 4^{\frac{n}{2}+1} S_{\frac{n}{2}+1} = AB + QM \quad (3.4)$$

Where $2^k 4^{\frac{n}{2}} = 2^{kn} = R$ and $Q = q_{\frac{n}{2}} \dots q_0$.

If we multiply (3.4) by R^{-1} we get

$$4S_{\frac{n}{2}+1} RR^{-1} = ABR^{-1} + QMR^{-1} \quad (3.5)$$

As $RR^{-1} \bmod M = 1$, then

$$RR^{-1} \bmod M = 1 + kM \quad (3.6)$$

and (3.5) becomes:

$$4S_{\frac{n}{2}+1} (1 + kM) = ABR^{-1} + QMR^{-1} \quad (3.7)$$

Completing the modular reduction we get:

$$ABR^{-1} \bmod M = 4S_{\frac{n}{2}+1} \bmod M \quad (3.8)$$

Because

$$S_{\frac{n}{2}+1} RR^{-1} = (AB + QM) / 2^k (4^{\frac{n}{2}+1}) \quad (3.9)$$

and $A, B < 2M$, then:

$$S_{\frac{n}{2}+1} = (AB + QM) / 2^k (4^{\frac{n}{2}+1}) < (4M^2 + QM) / 2^k (4^{\frac{n}{2}+1}) = (4M + Q)M / 2^k (4^{\frac{n}{2}+1}) \quad (3.10)$$

Because $AB < 4M^2$ and Q has n -bit length, the length of $M + Q/4$ is $n+1$ bits, thus:

$$S_{\frac{n}{2}+1} < 2^k (2^{n+1})M / 2^k (2^n) < 2M \quad (3.11)$$

Again, as with Algorithm 3.4 and Algorithm 3.5, the result of this algorithm is $S = A \cdot B \cdot R^{-1} \bmod M$. Thus two steps are needed to compute a general modular multiplication, with the result of the first step, S , assigned to B and with R^2 assigned to A in the second step as in (3.2).

The modifications used to simplify Blum's modified Montgomery modular multiplication algorithm (Algorithms 3.4 and 3.5) can be applied to our proposed algorithm (Algorithm 3.6) to eliminate the need for the last step. This gives the more efficient modular multiplication algorithm for computing $S = A \cdot B \cdot R^{-1} \bmod M$ as shown in Algorithm 3.7. This modified algorithm using consecutive binary digits a_{2i-1}, a_{2i} and a_{2i+1} is faster and more suited to hardware implementation.

Algorithm 3.7 Proposed high-radix Montgomery modular multiplication for hardware implementation

Input: $M = \sum_{i=1}^{n-1} (2^k)^i m_i, m_i \in \{0,1,\dots,2^k-1\},$
 $B = \sum_{i=1}^{n-1} (2^k)^i b_i, b_i \in \{0,1,\dots,2^k-1\},$
 $A = \sum_{i=1}^{n-1} (2^k)^i a_i, a_i \in \{0,1,\dots,2^k-1\}, a_{n+2} = 0,$
 $a_{n+3} = a_{n+2} = a_{n+1} = a_n = a_{-1} = 0,$
 $A, B < 2M, 4M < R = 2^{kn},$
 $M < R = 2^{kn}, M' = -M^{-1} \bmod 2^k, \gcd(2^k, M) = 1$
Output: $S = A \cdot B \cdot R^{-1} \bmod M$

1. $S_0 = 0$
 2. **For** $i=0$ **To** $(n+2)/2$
 - $q_i = (S_i) \bmod 2^k$
 - $S_{i+1} = (S_i + q_i M') / 2^k + (a_{2i-1} + a_i - 2a_{2i+1})B$
- EndFor**
-

3.5 Hardware Implementation

Algorithm 3.5 modifies Montgomery's algorithm into just a sequence of iterative add and shift operations. By reducing the number of iterations from n to $(n+2)/2 + 1$ our proposed algorithm achieves a reduction in the computation time by approximately half compared to the algorithm proposed in [Blum01]. For a 2048 bit RSA algorithm, each computation needs to add three 2048 bit operands, which, if added conventionally, could cause a large delay during the adding operation. Instead, we propose a pipelined architecture to compute modular exponentiation using an iterative add operation. This

pipelined “iterative-add” architecture allows us to improve the operation speed by avoiding the large time delay in the addition operations.

Our proposed architecture, based on Algorithm 3.7, is shown in Figure 3.1. The internal structure of a processing element (PE) is shown in Figure 3.2. The input a_i^* in Figure 3.1 and Figure 3.2 is $a_i^* = a_{2i-1} + a_{2i} - 2a_{2i+1}$. The major challenge for our architecture, as with any architecture for computing modular multiplication, is to improve the speed and avoid the large time delay for the addition operation. To achieve this performance improvement we use a carry save adder structure (as in Figure 3.2) to generate intermediate partial sum (TS^j) and partial carry (TC^j) terms, which are combined at each PE using a carry look-ahead adder (CLA) in the last l clock cycles. We see that from Figure 3.1:

$$B = B^{l-1}, B^{l-2}, \dots, B^1, B^0 \quad (3.12)$$

$$M = M^{l-1}, M^{l-2}, \dots, M^1, M^0 \quad (3.13)$$

$$S_i = S_i^{l-1}, S_i^{l-2}, \dots, S_i^1, S_i^0 \quad (3.14)$$

Where the chunks B^j, M^j, S_i^j are of the length w bits, with $l = \frac{n}{w}$ and $j \in \{0, 1, \dots, l\}$.

Then:

$$\begin{aligned} S_{i+1} = & ((S_i^0 + q_i M^0 + (a_{2i-1} + a_{2i} - 2a_{2i+1})B^0) \\ & + 2^w (S_i^1 + q_i M^1 + (a_{2i-1} + a_{2i} - 2a_{2i+1})B^1) \\ & + \dots + 2^{(l-1)w} (S_i^{l-1} + q_i M^{l-1} + (a_{2i-1} + a_{2i} - 2a_{2i+1})B^{l-1}) + 2^n S_i^n) / 2 \end{aligned} \quad (3.15)$$

$$\begin{aligned} S_{i+2} = & ((S_{i+1}^0 + q_{i+1} M^0 + (a_{2i+1} + a_{2i+2} - 2a_{2i+3})B^0) \\ & + 2^w (S_{i+1}^1 + q_{i+1} M^1 + (a_{2i+1} + a_{2i+2} - 2a_{2i+3})B^1) \\ & + \dots + 2^{(l-1)w} (S_{i+1}^{l-1} + q_{i+1} M^{l-1} + (a_{2i+1} + a_{2i+2} - 2a_{2i+3})B^{l-1}) + 2^n S_{i+1}^n) / 2 \end{aligned} \quad (3.16)$$

⋮

$$\begin{aligned} S_{\frac{n}{2}+1} = & ((S_{\frac{n}{2}}^0 + q_{\frac{n}{2}} M^0 + (a_{\frac{n}{2}-1} + a_{\frac{n}{2}} - 2a_{\frac{n}{2}+1})B^0) \\ & + 2^w (S_{\frac{n}{2}}^1 + q_{\frac{n}{2}} M^1 + (a_{\frac{n}{2}-1} + a_{\frac{n}{2}} - 2a_{\frac{n}{2}+1})B^1) \\ & + \dots + 2^{(l-1)w} (S_{\frac{n}{2}}^{l-1} + q_{\frac{n}{2}} M^{l-1} + (a_{\frac{n}{2}-1} + a_{\frac{n}{2}} - 2a_{\frac{n}{2}+1})B^{l-1}) + 2^n S_{\frac{n}{2}}^n) / 2 \end{aligned} \quad (3.17)$$

After the result S_{i+1} is calculated from (3.15), the least significant chunk S_{i+1}^0 is forwarded to the same PE during the next iteration as the first part of (3.16). Thus if we have l PEs to compute $S_i^j + q_i M^j + (a_{2i-1} + a_{2i} - 2a_{2i+1})B^j$, our pipelined architecture

can be applied as shown in Figure 3.3. For radix $k > 1$, the structure is similar to $k = 1$ where $q_i M^j$ and $(a_{2i-1} + a_{2i} - 2a_{2i+1})B^j$ are the results of the multiplication.

Figure 3.1 The high speed iterative-add architecture for modular multiplication

Figure 3.2 The internal structure of a processing element for the high speed iterative-add architecture

Based on the architecture proposed in Figure 3.1, our modular multiplication operates as follows:

- (1) Reset the TC_REG, B_REG and M_REG of each PE to 0;
- (2) Input $(a_{-1} + a_0 - 2a_1), (a_1 + a_2 - 2a_3), \dots, (a_{n-1} + a_n - 2a_{n+1})$ sequentially to PE_0 through the a_i port;
- (3) $(M^0, B^0), (M^1, B^1), \dots, (M^{l-1}, B^{l-1})$ are input from the M_BUS and B_BUS. Every PE computes the partial sum and partial carry (TS^j, TC^j) , where $j \in \{0, 1, \dots, l\}$, for the current iteration;
- (4) When PE_0 's input is $a_{n-1} + a_n - 2a_{n+1}$, the CSA adders start to compute the final result $S_{n/2+2}$;
- (5) After $(n/2 + 2 + l)$ computations, the result $S_{n/2+2}$ is placed on the S_BUS.

During the calculations, each PE contains the intermediate result S_i^j represented as the partial sum TS^j and the partial carry TC^j . These results are forwarded to the same PE for the calculation of S_{i+1}^j in the next cycle as shown in (3.15) - (3.17) and Figure 3.2 and 3.3. This avoids carry propagation in the adders, eliminating the need to use a CLA in each PE , achieving a 22% reduction in the computation time¹. Instead, we use a simpler iterative carry propagate in association with a w bit CLA, outside the PE , to generate the final result, as in Figure 3.1 and Figure 3.3, as:

Figure 3.3 The data flow diagram showing the flow of data through the proposed iterative-add architecture

¹ The 22% speed improvement is mainly due to the fact that the CSA adder is able to add three input digits consecutively, whereas the CLA adder can only add two input digits. Thus the CSA is more suited to the calculation of equations 3.15 to 3.17, where it can be seen that each sub-term involves two 3-digit additions.

$$S_{\frac{n}{2}+2} = FS_{\frac{n}{2}+1}^{l-1}, FS_{\frac{n}{2}+1}^{l-2}, \dots, FS_{\frac{n}{2}+1}^1, FS_{\frac{n}{2}+1}^0 \quad (3.18)$$

where

$$FS_{\frac{n}{2}+1}^j = TS_{\frac{n}{2}+1}^j + (TC_{\frac{n}{2}+1}^j \ll 1) + c_o^{j-1} \quad (3.19)$$

where TS^j and TC^j are the partial sum and partial carry for the j^{th} PE, respectively, $\ll$ represents at 1-bit left shift, and c_o^{j-1} is the carry out from the previous PE.

3.6 Performance Evaluation

3.6.1 Software Performance of our Proposed Algorithm

We have implemented Algorithm 3.7 in 'C' running on a 2.4GHz Pentium IV computer system running the microsoft windows XP operating system. Figure 3.4 shows the run times for modular multiplication for different radix (k) values and different key lengths. An implementation of Blum's algorithm takes 48.73ms for a 2048 bit modular multiplication, while our design takes 24.50 ms for the same computation, obtaining a 49.72% speed improvement. Comparing the results of Figure 3.4, we can see that the run time for our proposed algorithm is nearly half that of Blum's algorithm and as expected increases with increasing key size and radix value.

Figure 3.4 The run time (in ns) for (a) Blum's algorithm and (b) our proposed algorithm on a Pentium 4 (2.4 GHz) processor

3.6.2 Hardware Performance of our Proposed Algorithm

A hardware implementation of Algorithm 3.7 has been developed for FPGA using VHDL coding, Synplify synthesis, and an ALTERA EP20K1500EBC652-1 target. Figure 3.5 shows the total run time of our proposed algorithm compared to that of Blum's algorithm running on our target architecture. Again, as for the software version, the FPGA implementation of our proposed algorithm has a run time which is nearly half that of Blum's algorithm. To overcome the problem associated with the carry propagation delay, we use a 32-bit basic adder unit in the pipelined iterative-add architecture. This results in a run time for the FPGA implementation of $T = t_n \times n/2 + n/32$, where n is the key length and t_n is run time for different n . Thus, $2(n/2 + 2 + l)$ cycles are needed to accomplish one modular multiplication where $l = n/32$. Suppose we have an n -bit modular exponentiation computation, the total computation time is $(n + 1)(n/2 + 2 + l)$ cycles. Thus, our implementation provides a distinct advantage compared to other known algorithms. Comparing the simulation results of Figure 3.4 and Figure 3.5, the proposed hardware design is approximately 1000 times faster than the software version for large key lengths. As would be expected, the effective speedup for smaller key lengths is less impressive.

Figure 3.5 The run time (in ns) for (a) Blum's algorithm and (b) the proposed algorithm on FPGA

After the evaluation of our proposed architecture in FPGA, we deployed the design to an ASIC platform using the Atisan TSMC 0.18 μm standard cell library. The design is optimized for 2048 bit encryption, and unlike many other designs is scalable in that it is able to operate with a range of different key lengths and radix values (k). Because the design is scalable, as the key length increases more PEs are committed to the computation, but the clock frequency remains essentially the same. The total dynamic power consumption of our proposed design for a range of key lengths and radix values is given in Figure 3.6, while the performance results, along with comparisons of other designs from the literature, are summarized in Table 3.1.

Table 3.1 Modular exponentiation of hardware performance comparison

	Key Length	Freq (MHz)	MM (μs)	ME (ms)	Power (mW)	Platform
Kim[Kim04a]	1024 (2)	52	40.04*	40.05	-	Virtex XC40150XV
Tenca[Tenc03]	1024 (2,8)	80	43	45*	-	0.5 μm CMOS
Mcloone[Mclo04]	1024 (2)	95.9	10.65*	10.9*	-	XC2V6000
Kaihara[Kaih05]	512 (2)	114.16	3.02*	1.58*	-	0.35 μm CMOS
Chingwei[Chin06]	1024 (4)	460	1.71*	1.74*	830	0.18 μm CMOS
Nathaniel[Nath07]	1024 (4)	266	4.98*	5.1	-	XC2V2000-06
This Work	2048 (2^1)	305.8	71.3	14.6	406.3	0.18 μm CMOS
	1024 (2^1)		3.58	3.67	193.5	
This Work	2048 (2^2)	253.8	4.55	9.3	449.7	0.18 μm CMOS
	1024 (2^2)		2.28	2.34	205.4	
This Work	2048 (2^{32})	85.3	2.29	4.7	894.9	0.18 μm CMOS
	1024 (2^{32})		1.17	1.2	450	

MM = Modular Multiplication; ME = Modular Exponentiation;

* Author's estimation based on the original papers; - unable to be determined

Generally it is difficult to compare power consumption and processing speed with other published works as different parameters and different target technologies are used. From Table 3.1, our proposed method achieves reasonable performance for radix values ranging from 1 to 32. We achieve around 305.8 MHz for key lengths ranging from 128 bit to 2048 bit for a radix value $k = 1$. This drops off to around 85.3 MHz for key lengths ranging from 128 bit to 2048 bit for a radix value $k = 32$. The closest design in the literature is that of Chingwei *et al.* [Chin06], which achieved 586 kbit/s at a clock speed of 460 MHz for a 1024 bit RSA computation at radix $k = 2$. This results in around 1.71 μs for a 1024 modular multiplication and 1.74 ms for a 1024-bit full

modular exponentiation. Although our more general design, optimized for 2048 bit encryption, is 25% slower than [Chin06] for 1024 bit computations, our power consumption is about 1/4 of [Chin06]. A more useful measure for comparison purposes is the power-ME computation time product, which for our design is about 1/3 that of [Chin06].

Figure 3.6 The implementation's total dynamic power consumption (mW) for different key lengths on a 0.18 μ m CMOS ASIC

3.7 Conclusions

In this chapter, we have proposed an efficient algorithm for the modular multiplication which reduces the computation time by approximately half. The improved modular multiplication has been implemented both in FPGA and ASIC to examine the area-time characteristics for different radix values and key lengths. Our simulations show that it is superior to Blum's implementation for a range of key lengths and radix values. The ASIC version provides a performance which is as good as, if not better than, other implementations from the literature.

Chapter 4 An SD Adder based Unified Multiplier for Finite Fields $GF(p)$ and $GF(2^p)$

4.1 Introduction

The Rivest-Shamir-Adleman (RSA) encryption algorithm and the Elliptic Curve Cryptogram (ECC) dominate modern public key cryptosystems. Many architectures and hardware implementations have been proposed for performing RSA [Rive78, Koch85, Vand90, Royo97, Tenc99, Tenc01, Kits02, Kim04b] and ECC [Agne90, Agne93, Suti98a, Gao99, Orla00, Orla01, Moon01, Good01, Bati02, Erns03] encryption. Generally, these existing approaches for cryptographic processors only consider RSA or ECC individually. Unifying the two cryptosystems would achieve better performance, particularly in relation to power consumption and silicon area, by reusing hardware resources.

Some considerable research effort has already been expended on the efficient unification of modular multiplication, modular addition/subtraction and modular inversion over both $GF(p)$ and $GF(2^p)$ [Sava00, GroB01, Wolk02, Sato03, Gutu03, Bati04, Cila05, Crow05]. The difference between modular multiplication over $GF(p)$ and $GF(2^p)$ lies in the addition arithmetic, and in particular, whether the carry of the summand is forwarded or not. A modified full adder supporting carry propagation and carry-less modes of operation is commonly used in existing designs. However, this results in a critical path delay difference between the two modes of operation which affects the performance. For instance, in a conventional logic implementation of a full adder, the critical path has three simple gate delays for the carry and two XOR gate delays to produce the result. This contrasts to a critical path of just a single XOR gate delay for a carry-less mode of operation. This modified full adder approach has been applied in [Sava000, GroB01, Wolk02, Cila05] and [Crow05] to adopt the modular operations over both $GF(p)$ and $GF(2^p)$.

Most of the previously proposed methods rely on a carry-save adder to perform the operations of multiplication over the two fields: $GF(p)$ and $GF(2^p)$, allowing the reuse of the same hardware resources to support the operations of RSA and ECC. To overcome the critical path delay difference between the two modes of operation, we propose a novel public key cryptographic processor that is able to calculate all the necessary operations to support RSA and ECC over $GF(p)$ and $GF(2^p)$ fields. A dual-field arithmetic unit is developed to meet the above requirements including a dual-field addition/subtraction unit, a dual-field modular multiplication unit, and a dual-field

inversion unit. This arithmetic unit focuses on high-speed, low-cost modular multiplication, as this is the critical unit in the unified architecture. Central to this implementation is a new dual-field Signed Digit (SD) adder for supporting carry-propagation and carry-less modes of operation. The proposed adder only needs a two-level Look Up Table (LUT) to produce a carry, resulting in a significant reduction in the critical path and a corresponding improvement in the performance.

4.2 Preliminaries

In this section, we firstly introduce the SD number system and then proceed with a brief summary of RSA and ECC. Here, we will demonstrate that modular multiplication is the key to efficient implementation for both RSA and ECC.

4.2.1 The SD Number System

Using a conventional full adder results in a computation time which is dependant upon the three gate-delay critical path as shown in Figure 4.1 (a). Avizienis [Aviz61] proposed using a SD number implementation to avoid the carry operation so as to reduce the carry propagation time. A radix-2 (i.e. binary) SD number system consists of only the digit set $\{\bar{1}, 0, 1\}$, in which 1 and $\bar{1}$ in bit position i represent $+2^i$ and -2^i , respectively. The addition of two binary SD numbers x_i and y_i has been defined [Kore02] as follows:

$$u_i = (x_i + y_i) - 2c_{i+1} \quad c_{i+1} = \begin{cases} 1 & (x_i + y_i) \geq 1 \\ \bar{1} & (x_i + y_i) \leq \bar{1} \\ 0 & (x_i + y_i) = 0 \end{cases} \quad (4.1)$$

$$s_i = u_i + c_i \quad (4.2)$$

Where s_i, u_i, x_i, y_i and $c_{i+1} \in \{\bar{1}, 0, 1\}$. This results in a two stage operation to produce the final result. The first operation (Equation (4.1)) calculates the intermediate result, u_i , and the carry propagated to the next stage, c_{i+1} . While Equation (4.2) generates the final result, s_i , by adding u_i and c_i (the carry from the previous stage).

Figure 4.1 The structure of a full adder and an SD adder

The computation procedure for the possible bit patterns is shown in Table 4.1 and Table 4.2. These two steps are identical to the two LUTs of the SD adder shown in Figure 4.1 (b) and will be referred to as LUT1 and LUT2, respectively. This results in a carry propagation delay which is dependant on two LUTs, rather than the three gate delays in a conventional full adder. It should be noted that a timing analysis using Synopsys Design Compiler [Syno], targeting the $0.09 \mu\text{m}$ CMOS technology gives a propagation delay of 0.53 ns for a conventional full adder implementation and 0.28 ns for a single LUT. Thus the propagation delay for a single adder using LUTs (0.56 ns) is worse than for a single full adder. However when adding n -bit numbers, the delay using a conventional implementation is approximately $n * 0.28$ ns while the delay using an SD adder is just 0.56 ns.

Table 4.1 Rules for generating the carry (c_{i+1}) and sum (u_i) results for different inputs $x_i y_i$ in the SD number system

$x_i y_i$	00	01	10	$0\bar{1}$	$\bar{1}0$	11	$1\bar{1}$	$\bar{1}\bar{1}$
c_{i+1}	0	1	1	0	0	1	0	$\bar{1}$
u_i	0	$\bar{1}$	$\bar{1}$	$\bar{1}$	$\bar{1}$	0	0	0

Table 4.2 Rules for computing s_i from the intermediate result u_i and the carry c_i in the SD number system

$u_i c_i$	00	01	10	$0\bar{1}$	$\bar{1}0$	$1\bar{1}$
s_i	0	1	1	$\bar{1}$	$\bar{1}$	0

4.3 Algorithms for Implementing RSA and ECC

In this section, we give algorithms for implementing modular multiplication, modular addition/subtraction and modular inversion over $GF(p)$ and $GF(2^p)$. The definitions of RSA and ECC public key cryptosystems are given in chapter 2. Modular multiplication is needed for both the RSA implementation and the ECC point operations in the prime field. In addition, ECC needs point addition and point doubling operations, which also require modular addition/subtraction and modular inversion. An implementation to unify the above algorithms using a modified SD adder is described next.

4.3.1 Proposed Modular Multiplication over $GF(p)$ and $GF(2^p)$

The difference between modular multiplication over $GF(p)$ and $GF(2^p)$ is whether the carry is propagated or not. We propose a modified SD adder supporting both (carry propagate) addition and (no carry propagate) XOR functions to unify the required operations over $GF(p)$ and $GF(2^p)$, respectively. Therefore, these two different operations can share the same hardware resources in the implementation.

Many hardware algorithms for modular multiplication are based on Montgomery modular multiplication [Mont85]. In this method, in order to avoid the division operation, the two integers are transformed into m -residues and multiplied. The result of the m -residue multiplication is then transformed back into a normal representation. Eldridge and Walter [Eldr93] proposed a modified Montgomery modular multiplier for computing $A \cdot B \bmod M$. A simplified version for radix $r = 2$ is given in Algorithm 4.1, which shows that the modular multiplication is iteratively determined by the additions performed on lines 2.1, 2.2, 2.3 and 2.4 of Algorithm 4.1. The addition operation can be realized in a normal binary number system by using carry-save adders. Unfortunately, carry propagation in these designs is always a bottleneck, especially when operations on large numbers are performed.

Algorithm 4.1 A modified Montgomery's modular multiplication algorithm [Eldr93]

Input: $A = \sum_{i=0}^{n-1} 2^i a_i, a_i \in \{0,1\}, B = \sum_{i=0}^{n-1} 2^i b_i, b_i \in \{0,1\},$
 $M = \sum_{i=0}^{n-1} 2^i m_i, m_i \in \{0,1\}, A, B < M, M < R = 2^n,$
 $M' = -M^{-1} \bmod 2 = M, \gcd(2, M) = 1,$
Output: $A \cdot B \cdot R^{-1} \bmod M$

```

1.   $S_0 = 0;$ 
2.  For  $i = 0$  To  $n-1$  Do
2.1   $S_i = S_i + a_i B;$ 
2.2   $q_i = S_i \bmod 2;$ 
2.3   $P_i = S_i + q_i M;$ 
2.4   $S_{i+1} = P_i / 2;$ 
    EndFor
3.  If  $S_n \geq M$  Then  $S_n = S_n - M$  ;Else  $S_n$  ; EndIf

```

The intermediate result,

$$P_i = S_i + q_i M + a_i B \quad (P = \sum_{i=0}^n 2^i p_i, p_i \in \{0,1\}) \quad (4.3)$$

should be an even number otherwise the value of S_{i+1} after right shifting will be truncated giving an incorrect final result. We give a proof that P_i could not be an odd number during the computation in the above algorithm.

Lemma 1.: The intermediate result, $P_i = S_i + q_i M + a_i B$, ($P = \sum_{i=0}^n 2^i p_i, p_i \in \{0,1\}$), is an even number.

Proof: Firstly, if any of A, B and M are even, then P_i is obviously an even number. Thus, the following proof is only for the situation when A, B and M are all odd.

- (1) If $i=0$, $a_i B$ is an odd number and $q_0 = 1$. Thus P_0 is an even number,
- (2) If $i=1$, it is unknown if S_1 and a_1 are odd or even. Thus there are four possible combinations of S_1 and a_1 for the next computational step:
 - (2.1) if S_1 is an odd number and $a_1 = 0$,
then $q_2 = 1$ and P_2 is an even number;
 - (2.2) if S_1 is an odd number and $a_1 = 1$,
then $q_2 = 0$ and P_2 is an even number;
 - (2.3) if S_1 is a even number and $a_1 = 0$,
then $q_2 = 0$ and P_2 is an even number;
 - (2.4) if S_1 is a even number and $a_1 = 1$,
then $q_2 = 1$ and P_2 is an even number.

Therefore, based on mathematic induction, the intermediate value P_i will always be an even number for i ranging from 0 to $n-1$. (#End of Lemma 1)

Algorithm 4.2 The proposed modular multiplication algorithm over $GF(p)$ based on an SD adder

Input: $A = \sum_{i=0}^{n-1} 2^i a_i, a_i \in \{0,1\}, B = \sum_{i=0}^{n-1} 2^i b_i, b_i \in \{0,1\},$
 $M = \sum_{i=0}^{n-1} 2^i m_i, m_i \in \{0,1\}, A, B < M, M < R = 2^n,$
 $M' = -M^{-1} \bmod 2 = M, \gcd(2, M) = 1,$

Output: $A \cdot B \cdot R^{-1} \bmod M$

1. $\bar{A}, \bar{B}, \bar{M} = \text{convert}(A, B, M), \bar{S}_0 = 0;$
 2. **For** $i = 0$ **To** $n-1$ **Do**
 - 2.1 $\bar{S}_i = \text{SADD}(\bar{S}_i, \bar{a}_i \bar{B});$
 - 2.2 $\bar{q}_i = \bar{S}_i \bmod 2;$
 - 2.3 $\bar{P}_i = \text{SADD}(\bar{S}_i, \bar{q}_i \bar{M});$
 - 2.4 $\bar{S}_{i+1} = \bar{P}_i / 2;$
 3. $\bar{S}_n = \text{convert}(\bar{S}_n);$
 4. **If** $\bar{S}_n \geq M$ **Then** $\bar{S}_n = \bar{S}_n - M$ **Else** $\bar{S}_n$ **EndIf**
-

Based on the algorithm of Algorithm 4.1, we propose the modular multiplication algorithm, based on an SD number system, shown in Algorithm 4.2. Firstly, we transform the integers into a signed number representation which has the benefit of allowing carry-less computations. The function SADD() represents a modified SD addition which is explained in the next section, while $\bar{S}_i$ and $\bar{S}_{i+1}$ are SD numbers. Lastly, we need to convert $\bar{S}_n$ back to a conventional binary number format. The operations of lines 2.1, 2.2, 2.3 and 2.4 in the algorithm of Algorithm 4.1 can be interpreted as lines 2.1, 2.2, 2.3 and 2.4, respectively, in Algorithm 4.2. Line 2.2 gets the least significant bit (LSB) of $\bar{S}_i \pmod{2}$ while line 2.4 performs a 1-bit right shift of $\bar{P}_i$ (divide by 2). Lemma 1 shows that the intermediate product P_i will always be an even number, so it is safe to perform the 1-bit right shift.

Montgomery multiplication has been extended to finite field $GF(2^p)$ in [Koc98]. Multiplication over $GF(2^p)$ is achieved by multiplying the corresponding polynomials, $a(x)$, $b(x)$, by the modulo irreducible polynomial, $m(x)$. Multiplication over $GF(2^p)$ is simply the logic AND operation, while addition over $GF(2^p)$ is just the bit-wise XOR of the coefficients of the two polynomials. We have extended the algorithm of [Koc98] to support processing using SD numbers. Algorithm 4.3 shows our modified polynomial modular multiplication algorithm, where the function SXOR() represents the SD bit-wise XOR operation. Line 2.2 gets the LSB of $s(x)_i \pmod{x}$ while line 2.4 performs a 1-bit right shift of $p(x)_i$ (divide by x). Comparing the algorithms of Algorithm 4.2 and Algorithm 4.3 shows that only the main operation (SADD() for $GF(p)$ and SXOR() for $GF(2^p)$) is different, making it possible to use a single hardware arithmetic unit for computing these two algorithms.

Algorithm 4.3 The proposed modular multiplication algorithm over $GF(2^p)$ based on an SD adder

Input: $a(x) = \sum_{i=0}^{n-1} a_i x^i$, $a_i \in GF(2)$,
 $b(x) = \sum_{i=0}^{n-1} b_i x^i$, $b_i \in GF(2)$, $m(x) = x^n + \sum_{i=0}^{n-1} m_i x^i$, $m_i \in GF(2)$,
 $m(x)$ is an irreducible polynomial of degree n over the field $GF(2)$

Output: $a(x) \cdot b(x) \cdot x^{-n} \pmod{m(x)}$

1. $s(x)_0 = 0$;
2. For $i = 0$ To $n - 1$ Do
 - 2.1 $s(x)_i = \text{SXOR}(s(x)_i, a_i b(x)_i)$;
 - 2.2 $q_i = s(x)_i \pmod{x}$;

```

2.3       $p(x)_i = \text{SXOR}(s(x)_i, q_i m(x)_i);$ 
2.4       $s(x)_{i+1} = p(x)_i / x;$ 
      EndFor

```

4.3.2 Proposed Modular Addition and Subtraction over $GF(p)$ and $GF(2^p)$

We propose a modified SD adder supporting addition ($SADD()$) and subtraction ($SSUB()$) over $GF(p)$ and supporting addition/subtraction ($SXOR()$) over $GF(2^p)$. The addition of two integers, A and B , $0 < A, B < M$, is performed by calculating the sum, $(A + B) \bmod M$. Note here, A and B have the same bit length. If $(A+B)$ is larger than the modulo number M , the result is $(A+B-M)$, else it is $(A+B)$. The subtraction $(A - B) \bmod M$ is similar. If $(A-B)$ is less than zero, the result is $(A-B+M)$, else it is $(A-B)$. Based on these observations, the algorithms for modular addition and subtraction over $GF(p)$ are given in Algorithm 4.4.

Algorithm 4.4 The proposed modular addition and modular subtraction over $GF(p)$ based on an SD adder

Input: $M, 0 \leq A \leq M, 0 \leq B \leq M$

Output: $C = (A + B) \bmod M$

1. $C_1 = SADD(A, B);$
 2. $C_2 = SSUB(C_1, M);$
 3. If $C_2 < 0$ Then $C = C_1$; Else $C = C_2$; EndIf
-

Input: $M, 0 \leq A \leq M, 0 \leq B \leq M$

Output: $C = (A - B) \bmod M$

1. $C_1 = SSUB(A, B);$
 2. $C_2 = SADD(C_1, M);$
 3. If $C_1 < 0$ Then $C = C_2$; Else $C = C_1$; EndIf
-

Modular addition and subtraction over $GF(2^p)$ are performed by bit-wise XORing the coefficients of polynomials $a(x)$ and $b(x)$, where

$$a(x) = \sum_{i=0}^{n-1} a_i x^i, a_i \in GF(2) \quad (4.4)$$

$$b(x) = \sum_{i=0}^{n-1} b_i x^i, b_i \in GF(2) \quad (4.4)$$

And those operations have been shown in Algorithm 4.5.

Algorithm 4.5 The proposed modular addition and modular subtraction over $GF(2^p)$ based on an SD adder

Input: $p(x) = a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1}$, $a_i \in GF(2)$,

$q(x) = b_0 + b_1x + b_2x^2 + \dots + b_{n-1}x^{n-1}$, $b_i \in GF(2)$

Output: $f(x) = p(x) \pm q(x)$

$$f(x) = p(x) \pm q(x)$$

$$= (a_0 \pm b_0) + (a_1 \pm b_1)x + (a_2 \pm b_2)x^2 + \dots + (a_{n-1} \pm b_{n-1})x^{n-1}$$

4.3.3 Proposed Modular Inversion over $GF(p)$ and $GF(2^p)$

We also propose an SD adder with a one-bit shifter for supporting modular inversion and division over $GF(p)$ and $GF(2^p)$. This allows the direct calculation of $SADD(B, m)/2$ in the ELSE section of the IF statement at lines 2.1.2 and 2.2.2 of the modular inversion algorithm over $GF(p)$ of Algorithm 4.6 and of $SXOR(B, m)/x$ in the ELSE section of the **If** statement at line 2.1.2 of the modular inversion algorithm over $GF(2^p)$ in Algorithm 4.7.

The algorithms of Algorithm 4.6 and Algorithm 4.7 are used to compute the inverse a^{-1} of a positive element $a \in \{1, 2, \dots, m-1\}$, and are based on the binary extended gcd algorithm [Mene97]. The difference between our algorithms (Algorithm 4.6 and Algorithm 4.7) and the binary extended gcd algorithm of [Mene97] is that firstly we can make the assumption that the modulo number m is a random number and is thus odd, which results in a simplification of the algorithm which is detailed in [Mene97]. Secondly, the algorithms have been adapted for SD computations.

Algorithm 4.6 The proposed modular inversion algorithm over $GF(p)$ based on an SD adder

Input:	Prime number $m, a \in [1, m-1]$
Output:	$a^{-1} \bmod m$

1. $\mu = a, \nu = m, B = 1, D = 0;$
2. **While** $\mu \neq 0$ **Do**
 - 2.1. **While** $\mu_0 = 0$ **Do**
 - 2.1.1 $\mu = \mu / 2;$
 - 2.1.2 **If** $b_0 = 0$ **Then** $B = B / 2;$
Else $B = SADD(B, m) / 2;$ **EndIf**
 - EndWhile**
 - 2.2. **While** $\nu_0 = 0$ **Do**
 - 2.2.1 $\mu = \mu / 2;$
 - 2.2.2 **If** $d_0 = 0$ **Then** $D = D / 2;$
Else $D = SADD(D, m) / 2;$ **EndIf**
 - EndWhile**
 - 2.3. **If** $\mu \geq \nu$ **Then** $\mu = SSUB(\mu, \nu), B = SSUB(B, D);$
Else $\nu = SSUB(\nu, \mu); D = SSUB(D, B);$ **EndIf**
 - EndWhile**
3. **Return** $D.$

Algorithm 4.7 The proposed modular inversion algorithm over $GF(2^p)$ based on an SD adder

Input:	$m(x), a(x) \in GF(2^n)$
Output:	$a(x)^{-1} \bmod m(x)$

1. $\mu = a(x), \nu = m(x), B = 1, D = 0;$
2. **While** $\mu \neq 0$ **Do**
 - 2.1. **While** $\mu_0 = 0$ **Do**
 - 2.1.1 $\mu = \mu / x;$
 - 2.1.2 **If** $b_0 = 0$ **Then** $B = B / x;$
Else $B = SXOR(B, m) / x;$ **EndIf**
 - EndWhile**
 - EndWhile**
3. **If** $\mu = 1$ **Then** **Return** $B;$ **EndIf**
4. **If** $\deg(\mu) < \deg(\nu)$ **Then** $\mu \Leftrightarrow \nu, B \Leftrightarrow D;$ **EndIf**
5. $\mu = SXOR(\mu, \nu), B = SXOR(B, D);$
Goto step 2.

4.4 Hardware Implementation

In this section, we propose a unified architecture for a cryptographic coprocessor that supports RSA and ECC. The following operations, necessary for each algorithm's implementation, are supported:

- (1) RSA: Modular multiplication;
- (2) ECC: Point addition and point doubling over $GF(p)$;
- (3) ECC: Point addition and point doubling over $GF(2^p)$.

The operation core for the two fields involves just addition and XOR operations. There have been numerous methods reported for accelerating a full adder implementation and for extending it to support the XOR operation [GroB01, Wolk02, Cila05] and [Crow05]. These methods all use an extension to the full adder to support the XOR operation. However, the carry-less XOR operation proposed in these papers is still affected by the expensive carry propagation of the adder circuit. Our approach, based on the SD number system, uses a modified SD adder for both the addition and XOR operations. With this approach, we eliminate the carry propagation problem so that our proposed design can obtain a balance in performance between the carry propagation addition and the carry-less addition computations with no significant increase in hardware resources.

4.4.1 A Public Key Cryptographic Coprocessor

The structure of the proposed cryptographic coprocessor is shown in Figure 4.2. The proposed architecture includes an arithmetic unit, a signed number binary converter, an operation controller, special memory, registers files and a bus interface. The core unit of the proposed coprocessor is the arithmetic unit which can support dual-field modular operations including modular multiplication, modular addition/subtraction and modular inversion. The initial inputs are converted to SD format, all the intermediate results are represented in SD format and the final results must be converted back to a conventional binary format. The special memory module is to store encrypted secure information instead of sending it to external memory. The proposed architecture is suitable for use as a coprocessor in a System On Chip (SOC) solution, or as an integrated Intellectual Property (IP) module for general security applications.

The basic flow is as follows. The operations controller controls the general modular multiplication for the RSA cryptosystem and the modular addition, modular subtraction, modular multiplication and modular inversion for the ECC cryptosystem over $GF(p)$ and $GF(2^p)$, respectively. The data controller reads the operation data from

the special memory unit, transfers it to the arithmetic unit and writes the results back to the special memory unit. The register file temporarily stores the private key and session keys. If operations during one session are completed, the session keys are forcibly cleared. The special memory module does not store the secure information in sequence. So as to resist a simple physical attack, the data is inverted. When the data is fetched back into the arithmetic unit, it is again inverted back to its original state.

Figure 4.2 The structure for RSA and ECC operations

4.4.2 Unified Adder over $GF(p)$ and $GF(2^p)$

Modular addition and subtraction over $GF(p)$ are performed according to Algorithm 4.4, and require two different operations, SADD() and SSUB(). However, modular addition and subtraction over $GF(2^p)$ are performed by simply bit-wise XORing the coefficients of the two polynomials. Figure 4.3 shows the basic structure for computing modular addition and subtraction over $GF(p)$ and $GF(2^p)$. The dual field adder in Figure 4.3 (a) is configured to support addition and subtraction operations over $GF(p)$. In order to reuse the same arithmetic unit (SADDER), a subtraction operation is converted into an addition operation. The subtraction of two SD numbers

x_i and y_i is performed as $x_i + (-y_i)^2$, with the selection of addition and subtraction operations being controlled by the ADD/SUB signal. As described in Algorithm 4.4, several adding operations are performed. The proposed addition /subtraction structure would support the intermediate product (as in the calculation of $\bar{S}_i$ in line 2.3) and feed the intermediate result into x_i for the next operation. Figure 4.3 (b) shows the dual field adder unit configured for addition over $GF(2^p)$. Since addition and subtraction over $GF(2^p)$ are simply XOR operations, the y_i conversion is not required and y_i directly passes through the converter.

Figure 4.3 Dual-field modular addition and subtraction

4.4.3 A Unified Modular Multiplier over $GF(p)$ and $GF(2^p)$

We use a 32-bit computation unit as the basic unit to implement the dual-field modular multiplication algorithms outlined in Algorithm 4.2 and Algorithm 4.3. To achieve the n -bit values for B , M and S_i from Algorithm 4.2 are segmented into l 32-bit components as:

$$\begin{aligned}
 \bar{B} &= \bar{B}^{l-1}, \bar{B}^{l-2}, \dots, \bar{B}^1, \bar{B}^0 \\
 \bar{M} &= \bar{M}^{l-1}, \bar{M}^{l-2}, \dots, \bar{M}^1, \bar{M}^0 \\
 \bar{S}_i &= \bar{S}_i^{l-1}, \bar{S}_i^{l-2}, \dots, \bar{S}_i^1, \bar{S}_i^0
 \end{aligned} \tag{4.5}$$

where $\bar{B}^k, \bar{M}^k, \bar{S}_i^k$ are SD values of length 32-bits ($k \in \{0, 1 \dots l-1\}$) and $l = \left\lceil \frac{n}{32} \right\rceil$. The parameters of Algorithm 4.3 are segmented in a similar way.

² To perform the subtraction as an addition we must convert y_i into $\bar{y}_i$. In an SD number system, the conversion is fast as we only need to multiply each bit by -1. For example, when computing $x - y$ ($x = 1010, y = 0101$), we convert y into $\bar{y} = 0\bar{1}0\bar{1}$, and then add $x + \bar{y}$ to give $x - y = 1010 + 0\bar{1}0\bar{1} = 1\bar{1}1\bar{1} = 0101$.

Modular multiplication over $GF(p)$ needs to compute the carry and forward it to the next level while modular multiplication over $GF(2^p)$ is a carry-less computation. Thus the two computations for modular multiplication mainly involve addition and XOR operations. Figure 4.4 shows the structure of the 32-bit dual-field modular multiplier which is based on the dual-field signed adder unit described previously. The selector, D , either forwards the carry to the next level or not according to the field selection. The parameter $\bar{a}_i$ is the SD representation of an element a_i in $A = \sum_{i=0}^{n-1} 2^i a_i, a_i \in \{0,1\}$ and $\bar{q}_i$ is a 2-bit SD intermediate result. The inputs $\bar{b}_{z0...31}$, $(\bar{s}_i)_{z0...31}$ and $\bar{m}_{z0...31}$ correspond to the terms $\bar{b}_{32k+j}$, $(\bar{s}_i)_{32k+j}$ and $\bar{m}_{32k+j}$ respectively, as in Equation (4.6):

$$\begin{aligned}\bar{B}^k &= \sum_{j=0}^{31} 2^{32k+j} \bar{b}_{32k+j} = \sum_{j=0}^{31} 2^{zj} \bar{b}_{zj} \\ \bar{M}^k &= \sum_{j=0}^{31} 2^{32k+j} \bar{m}_{32k+j} = \sum_{j=0}^{31} 2^{zj} \bar{m}_{zj} \\ \bar{S}_i^k &= \sum_{j=0}^{31} 2^{32k+j} (\bar{s}_i^k)_{32k+j} = \sum_{j=0}^{31} 2^{zj} (\bar{s}_i^k)_{zj}\end{aligned}\quad (4.6)$$

where $j \in \{0,1,...,31\}$ and $z = "32k+"$.

Figure 4.4 The implementation of dual-field modular multiplication

4.4.4 Unified Modular Inversion over $GF(p)$ and $GF(2^p)$

The operations of modular inversion over $GF(p)$ and $GF(2^p)$ are similar to modular addition and subtraction over $GF(p)$ and $GF(2^p)$. In Algorithm 4.6 and Algorithm 4.7,

the basic operation of modular inversion is modular addition followed by a 1-bit right shift. Figure 4.5 shows the structure of an SD adder with a 1-bit right shifter. Figure 4.5(a) shows the improved adder unit configured for modular inversion over $GF(p)$. The 1-bit right shift is selected by the *Shift* signal. Figure 4.5(b) shows the improved adder unit configured for modular inversion over $GF(2^p)$.

Figure 4.5 The implementation of the dual-field SD adder-shifter

4.4.5 Special Memory

Memory is one of the most important modules in the proposed architecture due to the need for secure information storage. In order to resist against physical attacks, we design a special memory unit for locally storing the secret information, such as the private key. The memory is also forcibly flushed, by overwriting with a random number, to destroy the current session key when a session operation terminates. Information stored in memory is stored in 32-bit word reverse order to resist a simple probing attack. That is if the private key is given by:

$$\underbrace{k_{31}^j k_{30}^j \dots k_1^j k_0^j}_{32} \underbrace{k_{31}^{j-1} k_{30}^{j-1} \dots k_1^{j-1} k_0^{j-1}}_{32} \dots \underbrace{k_{31}^0 k_{30}^0 \dots k_1^0 k_0^0}_{32}, \text{ where } j = \{64, 32, 16, 8\} \quad (4.7)$$

Then, the key is stored in memory as:

$$\underbrace{k_0^j k_1^j \dots k_{30}^j k_{31}^j}_{32} \underbrace{k_0^{j-1} k_1^{j-1} \dots k_{30}^{j-1} k_{31}^{j-1}}_{32} \dots \underbrace{k_0^0 k_1^0 \dots k_{30}^0 k_{31}^0}_{32}, \text{ where } j = \{64, 32, 16, 8\} \quad (4.8)$$

4.4.6 Performance Evaluation

The critical path of our proposed design lies in the modular multiplication operation. Therefore, it is important to speed up the dual-field modular multiplier. This architecture can support up to $32 \times l$ bit operations for RSA and ECC cryptosystems if the structure has a 32-bit datapath for data transformation. Even if carry propagation is avoided within one unit, the carries produced among the units can not be ignored when computing modular exponentiation over $GF(p)$.

We propose a pipelined architecture to compute modular exponentiation using an iterative add operation. This pipelined "iterative add" architecture allows us to improve the operation speed by avoiding the large time delay in the addition operations detailed in the algorithm of Algorithm 4.2. This iterative procedure is shown below with $\bar{B}$, $\bar{M}$ and $\bar{S}_i$ as defined in Equation (4.5):

$$\begin{aligned} \bar{S}_{i+1} = & \{(\bar{S}_i^0 + \bar{q}_i \bar{M}^0 + \bar{a}_i \bar{B}^0) + 2^{32}(\bar{S}_i^1 + \bar{q}_i \bar{M}^1 + \bar{a}_i \bar{B}^1) + \dots + \\ & + 2^{(l-1)32}(\bar{S}_i^{l-1} + \bar{q}_i \bar{M}^{l-1} + \bar{a}_i \bar{B}^{l-1})\} / 2 \end{aligned} \quad (4.9)$$

In the next iteration:

$$\begin{aligned} \bar{S}_{i+2} = & \{(\bar{S}_{i+1}^0 + \bar{q}_{i+1} \bar{M}^0 + \bar{a}_{i+1} \bar{B}^0) + 2^{32}(\bar{S}_{i+1}^1 + \bar{q}_{i+1} \bar{M}^1 + \bar{a}_{i+1} \bar{B}^1) + \dots + \\ & + 2^{(l-1)32}(\bar{S}_{i+1}^{l-1} + \bar{q}_{i+1} \bar{M}^{l-1} + \bar{a}_{i+1} \bar{B}^{l-1})\} / 2 \end{aligned} \quad (4.10)$$

As can be seen from Equations (4.9) and (4.10), the lower 32-bit result $\bar{S}_{i+1}^0$ from Equation (4.9) is forwarded to the first term in Equation (4.10). Thus if a single computational unit k computes $\bar{S}_i^k + \bar{q}_i \bar{M}^k + \bar{a}_i \bar{B}^k$, then by pipelining l of these units the final result is calculated. This proposed architecture is given in Figure 4.6 and has l pipeline stages to compute the modular operation. $UNIT_l$ could execute the computation upon receiving the result from $UNIT_0$, while $UNIT_0$ is computing the second iteration of the modular multiplication. Using this l stage pipeline architecture, $2(n+1)+l$ cycles are needed to accomplish one modular multiplication. Suppose we have an n -bit modular exponentiation computation, the total computation time is $(n+1)(2n+2)+l$ cycles. The above computation is applied to modular multiplication over $GF(2^p)$ without carry propagation, by forcing the carry propagation between any two units to zero.

4.5 Comparisons and Results

We used VHDL coding, Synplify synthesis, Xilinx Virtex-IV XC4VFX100 FF1517-10, Virtex-E XCV2000E BG560-6 and XC40150XV BG560-7 FPGA devices and a 0.09 μm CMOS standard cell technology to implement the proposed design. Table 4.3 shows the performances of the proposed architecture on these different FPGA

platforms. As we are unable to implement 3-level logic in the FPGA devices, we have encoded the SD numbers as $\{0=00; 1=01; \bar{1}=10\}$. This has the effect of doubling the datapath area required compared to a conventional binary number scheme. All the tables listed here refer to the key size of 1024 bits.

Figure 4.6 A fast dual-field modular multiplication unit

Table 4.4 shows the performance of the proposed design under different timing constraints using a $0.09 \mu\text{m}$ CMOS standard cell technology. Here we see that the multiplier unit is the limiting module in terms of both silicon area and operating speed. It is also the module that consumes the most power.

Table 4.3 The performance of the proposed design on Virtex-4 XC4VFX100, Virtex-E XCV2000E and Virtex XC40150XV FPGA devices

	XC4VFX100-10		XVC2000E-6		XC40150XV-7	
The Proposed Work	Area CLBs	Freq MHZ	Area CLBs	Freq MHZ	Area CLBs	Freq MHZ
Multiplier(1024)	2793	150.5	4355	100.4	8399	44.4
Adder(193)	293	314.8	587	202.9	798	113.6
Inversion(193)	1788	150.8	2433	100.5	4286	45.8
Controller	353	-	728	-	867	-
Total	5227	150.5	8103	100.4	14290	44.4

Table 4.5 shows a hardware performance comparison of existing designs for modular multiplication and modular exponentiation of RSA and ECC over $GF(p)$ and $GF(2^p)$.

This table compares operating frequency, computation times and area for a number of published designs with different parameters and different target technologies. While it is difficult to make exact comparisons, our unified architecture performs reasonable well with an operating frequency of 44.2 MHz using 8339 CLBs (see Table 4.5) when compared to the highly optimized RSA only reconfigurable high-radix modular multiplication and exponentiation implementation of Blum *et al.* [Blum01] (52 MHz with 4865 CLBs with a 1024-bit key length and a radix value of 2 as shown in Table 4.5). Both designs target the same (superseded) Xilinx XC40150XV FPGA. As mentioned above, our SD implementation requires double the hardware area (CLBs) due to the necessity for coding the SD digits in the FPGA implementation.

Table 4.4 The performances of the proposed design for different timing constraints on 0.09 μ m CMOS standard cell technology

	Timing Constraint	5ns	1ns	0.5ns
Multiplier	Area(Gates)	362K	436K	505K
	Power(W)	0.0548	0.126	1.297
	Freq(MHZ)	215.1	1087.1	1351.4
Inverse	Area(Gates)	160K	257K	313K
	Power(W)	0.0114	0.0725	0.134
	Freq(MHZ)	252.5	1089.2	2381.0
Adder	Area(Gates)	60K	63K	68K
	Power(W)	0.0102	0.0494	0.1131
	Freq(MHZ)	689.7	1111.1	2272.7

Table 4.5 Multiplier hardware performance comparison

RSA	Key Size (bits)	Freq (MHZ)	MM (μ s)	ME (ms)	Area	Platform
Royo	768(2)	50	8	10	77 mm^2	ES2 0.7 μ m CMOS
Tenca	1024(2)	90	34.18	36*	1699 mm^2	1.2 μ m CMOS
Blum	1024(2)	52	40.04 *	40.05	4865 CLBs	Virtex XC40150XV
Tenca	1024(2,8)	80	43	45*	Unknown	0.5 μ m CMOS
ECC	Field	Freq (MHZ)	PM (ms)	PD+PA (ms)	Area	Platform
Okada	$GF(2^{163})$	3	8	-	Unknown	EPF10K250AGC599-2
	$GF(2^{163})$	66	1.1	-	165K gates	0.25 μ m CMOS
Orlando	$GF(2^{167})$	76.7	0.21	0.36	3002 LUTs	XCV400E-8-BG432
Orlando	$GF(p)$ 192bits	40	3	-	11416 LUTs	XCV1000E-8-BG680
Goodman	$GF(2^{1024})$	50	32.1	-	Unknown	0.25 μ m CMOS

MM=Modular Multiplication; ME=Modular Exponentiation;

PM=Point Multiplication; PA=Point Adding; PD=Point Doubling;

* Author's estimate based on the original papers; -Not supported.

It is difficult to compare current ECC implementations due to the different parameters that are chosen in each implementation. We chose a 192-bit ECC implementation over $GF(p)$ and a $GF(2^{163})$ ECC implementation over $GF(2^p)$ and achieved a clock speed of 100.4 MHz on a Xilinx Virtex-E (XVC2000E) FPGA platform. This is more than 2.5 times faster than Orlando's [Orla01] 40 MHz, 2854 CLBs implementation shown in Table 4.5 for the same target family.

Table 4.6 Unified hardware performance comparison

ECC & RSA	Field	Freq (MHZ)	ME (ms)	PM (ms)	Area	Platform
Savas	RSA1024bits	90	27.03*	-	Unknown	1.2 μ m CMOS
	$GF(p)$ 160bits	90	0.67*	-	Unknow	1.2 μ m CMOS
Satoh	$GF(p)$ 192bits	137.7	1.44	-	11.3K gates	0.13 μ m CMOS
	$GF(2^{160})$	510.2	-	0.19	10.8K gates	0.13 μ m CMOS
Batina	RSA1024bits	53	22.8	-	Unknown	Virtex 800
	$GF(2^{275})$	53	11.4	-	Unknown	Virtex 800
Crowe	RSA1033bits	44.91	23.8*	-	5267 slices	Virtex-II XC2V200
	$GF(2^{256})$	44.91	1.47*	-	5267 slices	Virtex-II XC2V200
Cilardo	RSA1024bits	77.34	27.36	-	6709 CLBs	Virtex-E XVC2000E
	$GF(2^{160})$	77.34	-	1.18*	6709 CLBs	Virtex-E XVC2000E
This Work	RSA1024bits	150.5	13.94	-	5227 CLBs	Virtex-IV C4VFX100
	$GF(p)$ 192bits	150.5	0.542	-	5227 CLBs	Virtex-IV C4VFX100
	$GF(2^{163})$	150.5	-	0.347	5227 CLBs	Virtex-IV C4VFX100
This Work	RSA1024bits	100.4	21.09	-	8103 CLBs	Virtex XVC2000E
	$GF(p)$ 192bits	100.4	0.813	-	8103 CLBs	Virtex XVC2000E
	$GF(2^{163})$	100.4	-	0.520	8103 CLBs	Virtex XVC2000E
This Work	RSA1024bits	44.4	47.25	-	14290 CLBs	Xilinx XC40150XV
	$GF(p)$ 192bits	44.4	1.83	-	14290 CLBs	Xilinx XC40150XV
	$GF(2^{163})$	44.4	-	1.175	14290 CLBs	Xilinx XC40150XV
This Work	RSA1024bits	1087.1	1.93	-	437K gates	0.09 μ m CMOS
	$GF(p)$ 192bits	1087.1	0.0751	-	437K gates	0.09 μ m CMOS
	$GF(2^{163})$	1087.1	-	0.048	437K gates	0.09 μ m CMOS

MM=Modular Multiplication; ME=Modular Exponentiation;

PM=Point Multiplication; PA=Point Adding; PD=Point Doubling;

* Author's estimate based on the original papers; -Not supported.

Table 4.6 shows the latest unified implementations of RSA and ECC and compares operating frequency, computation times and area for a number of published designs and for our proposed design. The different designs listed can not be compared directly because of the different parameters and technologies employed. To overcome this we have targeted our design to a number of different technologies so that some form of comparison can be conducted. Our proposed modular design can achieve a clock frequency of 1 GHz using 776K gates using 0.09 μ m CMOS standard cell technology. The critical path of the proposed design is only two levels of LUT. Orlando and Paar

[Orla00] achieved 0.21 ms for point multiplication over $GF(2^{167})$. But this architecture is optimized for the field using $P(x)=x^{167}+x^6+1$ which lacks flexibility when compared with our proposed architecture. Satoh and Takana [Sato03] achieve 0.19 ms at 510.2 MHz for ECC over $GF(2^{160})$ and 1.44 ms at 137.7 MHz for 192 bit ECC over $GF(p)$. If we combine the two fields together, the maximum frequency is 137.7 MHz and the operation time for ECC over $GF(2^{160})$ will be 0.70 ms. With the trade-off of speed, unification and hardware resources, our proposed design is the fastest and most efficient.

4.6 Conclusion

We have proposed a cryptographic coprocessor that supports both RSA and ECC over $GF(p)$ and $GF(2^p)$. The supported operations include modular multiplication, point adding, and point doubling. An efficient modified SD adder has been proposed to address the unification of carry propagation and carry-less operations, while avoiding the critical path delay difference between the two modes. A dual-field arithmetic unit has been developed to implement modular addition/subtraction, modular multiplication and modular inversion over both fields. Pipelining is used to implement the proposed design to reduce computation time. The critical path of the proposed design includes only two levels of LUT and some selection logic. The proposed design is ported to FPGA to prove the efficiency and later to ASIC to demonstrate the performance. Our analysis shows that the unified design can operate at 1 GHz using 776K gates in a 0.09 μm CMOS standard cell technology.

Chapter 5 Countermeasures for Power Analysis Attacks

5.1 Introduction

In this chapter, we introduce a relatively new threat to hardware-based cryptographic implementations: The power analysis attack. A power analysis attack is an efficient way to break general RSA and ECC encryption implementations in hardware. We also present a Differential Power Analysis (DPA) attack using an initial value, called an “initial value DPA” attack, which can compromise Kim's countermeasure [Kim05] against power analysis attacks. We then propose a novel countermeasure which provides a better defense against both Simple Power Analysis (SPA) and DPA attacks.

Side channel attacks are a powerful method to recover sensitive data from smart cards or other physical devices by using timing, power and fault analysis, etc. In 1999, Kocher *et al.* [Koch99] gave an outline of power analysis including SPA and DPA. Many power attacks to elliptic curve cryptosystems were discussed such as Refined Power Analysis (RPA) and Zero Power Analysis (ZPA), both of which use the special features of the elliptic curve where the power consumption is zero. RSA cryptosystems use a private key as the exponent which if unprotected, can be determined relatively easily by means of power analysis, thus both RSA and ECC suffer from DPA attacks [Boer03, Okey00].

A number of countermeasures have been proposed to defend against the above power analysis attacks [Coro99, Fisc02, Izu03, Kim05, Mami04, Moll01, Kim04b]. Most countermeasures use randomization of the private key or blinding of the point P to defend against power analysis attacks. Unfortunately, the computation complexity of these countermeasures is increased significantly compared to the general algorithm. What is needed is a balanced approach to the algorithm's computational cost and resistance to power attacks.

In this chapter, a new power analysis attack, called an “initial value DPA” attack, is proposed. We also propose a novel countermeasure against “initial value DPA”, DPA and SPA attacks for ECC and show how this countermeasure can be extended to RSA. Finally, a dynamic power analysis of the proposed algorithm shows that it is superior to other techniques for minimizing power variations, thus making it less susceptible to power attacks.

5.2 Power Attack Countermeasures for ECC

To compute the scalar multiplication of d and P , the following generic techniques have been proposed. Algorithm 5.1 shows the binary method proposed by Satoh and Araki [Sato98], while Algorithm 5.2 shows Montgomery's method [Mont87].

Algorithm 5.1 Binary method (MSB) [Sato98]

Input: e, P

Output: $Q = eP$

1. $Q = P$
 2. **For** $i = n-2$ **Down to** 0
 - $Q = 2Q$
 - If** $e_i = 1$ **Then**
 - $Q = Q + P$
 - EndIf**
 - EndFor**
 3. **Return** Q
-

Algorithm 5.2 Montgomery's method [Mont87]

Input: e, P

Output: $Q = eP$

1. $Q_0 = P$
 2. $Q_1 = 2P$
 3. **For** $i = n-2$ **Down to** 0
 - $Q_{1-e_i} = Q_0 + Q_1$
 - $Q_{e_i} = 2Q_e + Q_1$
 - EndFor**
 4. **Return** Q
-

5.2.1 Mamiya's Countermeasure for ECC

Mamiya [Mami04] provided general countermeasures against RPA, ZPA, by ensuring that the power consumption during the computation was randomized. This meant that any special point R or a zero valued register does not appear during computation. Therefore, this countermeasure was shown to resist against ZPA, RPA, SPA and DPA for ECC public key cryptosystems [Mami04]. Algorithm 5.3 shows Mamiya's countermeasure for a power analysis attack of ECC.

Algorithm 5.3 Mamiya's method to compute eP for ECC**Input:** e, P **Output:** $Q = eP$ Choose a random elliptic point R

$$T_0 = R, T_1 = -R, T_2 = P - R$$

1. **For** $i = n-1$ **Downto** 0

$$T_0 = 2T_0$$

If $e_i = 0$ **Then**

$$T_0 = T_0 + T_1$$

Else

$$T_0 = T_0 + T_2$$

EndIf**EndFor**2. **Return**($Q = T_0 + T_1$)**5.2.2 Kim's Countermeasure**

Kim also presented a countermeasure against RPA, ZPA, SPA and DPA for ECC public key cryptosystems [Kim05] which avoided the computation of the negative random point R . This countermeasure is presented as Algorithm 5.4.

Algorithm 5.4 Kim's method to compute eP for ECC**Input:** e, P **Output:** $Q = eP$

Pre-computation

 $s = \#\mathcal{E} - e$, where $\#\mathcal{E}$ is the number of points of the curvesChoose a random elliptic point R

$$T[00] = \partial, T[01] = R, T[10] = P + R, T[11] = P + 2R$$

1. $Q = T[00]$ 2. **For** $i = n-1$ **Downto** 0

$$Q = 2Q$$

$$Q = Q + T[e_i s_i]$$

EndFor3. **Return** ($Q = eP$)**5.2.3 The Weakness of Kim's Countermeasure for ECC**

It is still possible to perform a power analysis to break Kim's countermeasure. The initial value $T[00] = \partial$ in Kim's algorithm provides the mechanism for the possible

attack. Power is consumed when the register values change, therefore, when a variable is constant, less power is consumed. This results in the power consumption during the calculation at $T[00] = \partial$ having a lower and constant value, which effects the power consumption of the complete circuit. Suppose the attackers know the highest bits $e_{n-1}, \dots, e_{i+1}$ of the secret e in Algorithm 5.4. For any given input P , the value of Q at the end of the i^{th} step of the loop is:

$$Q = yP + zR + e_i P + (e_i + s_i)R \quad (5.1)$$

Where $y = \sum_{j=i+1}^{n-1} e_j 2^{j-i}$ and $z = \sum_{j=i+1}^{n-1} e_j 2^{j-i} + \sum_{j=i+1}^{n-1} s_j 2^{j-i}$

The values of Q during the $(i+1)^{th}$ step have four possible cases:

$$\text{If } e_i = 0 \text{ and } s_i = 0, \text{ then } Q = 2yP + 2zR \quad (5.2)$$

$$\text{If } e_i = 0 \text{ and } s_i = 1, \text{ then } Q = 2yP + 2zR + R \quad (5.3)$$

$$\text{If } e_i = 1 \text{ and } s_i = 0, \text{ then } Q = 2yP + 2zR + P + R \quad (5.4)$$

$$\text{If } e_i = 1 \text{ and } s_i = 1, \text{ then } Q = 2yP + 2zR + 2P + R \quad (5.5)$$

From the above equations, the power consumed during the calculation of (5.2) will be different from, and less than, the other three cases. This is particularly the case when an iterative procedure for the calculation of the modular multiplication of ECC is used. Therefore, the initial values of Kim's countermeasures are a possible weak point which can lead to the leaking of sensitive information. This attack, called an "initial value DPA" attack could be countered if the initial value $T[00]$ was assigned a random value instead of the value ∂ .

5.2.4 Proposed Countermeasure against DPA for ECC

The possible attack to Kim's countermeasure can be avoided when the initial value $T[00]$ is assigned the value $\# \mathcal{E}R$, where $\# \mathcal{E}$ is the order of the curve (The order of the curve is the number of distinct points on the curve, including the zero point), instead of ∂ as in Algorithm 5.5. Even if the attackers know the highest bits $e_{n-1}, \dots, e_{i+1}$ of the secret key e , the power consumption during the $(i+1)^{th}$ step when computing C varies even when $s_i = 0$ and $e_i = 0$. Then:

$$eP = eP + y\# \mathcal{E}R \quad (5.6)$$

where the term $\# \mathcal{E}R$ is equal to a point at infinity. Thus $y\# \mathcal{E}R$ is also equal to a point at infinity, and hence $Q = eP + y\# \mathcal{E}R = eP$.

Algorithm 5.5 Our proposed method to compute eP for ECC**Input:** e, P **Output:** $Q = eP$

Pre-computation

 $s = \#\mathcal{E} - e$, where $\#\mathcal{E}$ is the number of points on the curveChoose a random elliptic point R $T[00] = \#\mathcal{E}R$, $T[01] = R$, $T[10] = P + R$, $T[11] = P + 2R$

1. $Q = T[00]$
2. **For** $i = n-1$ **Downto** 0
 - $Q = 2Q$
 - $Q = Q + T[e_i s_i]$
- EndFor**
3. **Return** ($Q = eP$)

Our proposed modified countermeasure avoids the weak points of Kim's countermeasure without significantly increasing the computational complexity. It also has the advantage of avoiding the computation of the inverse modular exponentiation of the random number r and the subtraction operation of the random elliptic point R . Compared with Kim's algorithm, our algorithm is a more efficient method to defend against SPA and DPA and provides a cost-effective method for computing modular exponentiation. Although there is a pre-computation step that is needed for our proposed algorithm, this is computed only once and thus the computation cost is not significant compared with that of modular exponentiation, which is iterated many times.

5.3 Power Attack Countermeasures for RSA

Power analysis attacks, and in particular SPA, are efficient methods to break general modular exponentiation computations based on a secret key e because of the zero power consumption when $e_i = 0$. Algorithm 3.1 in Chapter 3 is the proposed method to defend against a timing attack, the proposed algorithm can also defend against SPA. Algorithm 3.1, repeated here as Algorithm 5.6, shows the countermeasure against SPA for RSA. By adding some time lag to resist a linear timing attack and SPA, the time and power consumption for calculating the algorithm is uniformly independent of the value of e_i .

Algorithm 5.6 The proposed countermeasure against SPA for RSA**Input:** d, e **Output:** $C = d^e \bmod M$

```

 $d = 1$ 
1. For  $i = n-1$  Downto 0 Do
     $C = d \cdot d \bmod M$ 
2.   If  $e_i = 1$  Then
        $C = d \cdot m \bmod M$ 
   Else
        $C = d \cdot r \bmod M$ 
   EndIf
EndFor

```

5.3.1 Mamiya's Countermeasure

Mamiya's method can be further extended to RSA. Algorithm 5.7 shows the modified countermeasure against DPA for RSA based on Mamiya's method. It is obvious that the computation cost of the modified Mamiya's countermeasure is significantly increased by the modular inversion operation.

Algorithm 5.7 Mamiya's method to compute $d^e \bmod M$ for RSA**Input:** d, e **Output:** $C = d^e \bmod M$

```

Pre-computation
Choose a random number  $r$ ,
 $T_0 = r, T_1 = r^{-1}, T_2 = d \cdot r^{-1}$ 
1.  $C = T_0$ 
2. For  $i = n-1$  Downto 0
     $C = C^2 \bmod M$ 
    If  $e_i = 0$  Then
         $C = C \cdot T_1 \bmod M$ 
    Else
         $C = C \cdot T_2 \bmod M$ 
    EndIf
EndFor
3. Return( $C = C \cdot r^{-1}$ )

```

5.3.2 Kim's Countermeasure

Kim *et al.* [Kim05] proposed an improved countermeasure based on Mamiya's method which avoided inversion of the modular exponentiation computation of the random number r for RSA public key cryptosystems. Kim's countermeasure against SPA and DPA for RSA public key cryptosystems is presented as Algorithm 5.8.

Algorithm 5.8 Kim's method to compute d^e for RSA

Input: d, e

Output: $C = d^e \bmod M$

Pre-computation

$s = \phi(M) - e$, where $\phi(M)$ is Euler phi-function

Choose a random number r ,

$T[00] = 1, T[01] = r, T[10] = d \cdot r, T[11] = d \cdot r^2$

1. $C = T[00]$
 2. **For** $i = n-1$ **Downto** 0
 - $C = C^2 \bmod M$
 - $C = C \cdot T[e_i s_i] \bmod M$
 - EndFor**
 3. **Return**($C = d^e \bmod M$)
-

5.3.3 Weakness of Kim's Countermeasure for RSA

While Kim's countermeasure is an improvement over the basic modular exponentiation algorithm implementation, it is still possible to perform a power analysis to break Kim's countermeasure. As with the ECC case, the initial value $T[00] = 1$ provides the mechanism for the possible attack. Power is consumed when the register values change, therefore, when a variable is constant, less power is consumed. In the calculation of C , as in Algorithm 5.8, when $T[00] = 1$, the calculation has a constant value which effects the power consumption of the circuit.

For instance, if an attacker again knows the highest bits $e_{n-1}, \dots, e_{i+1}$ of the secret e in Algorithm 5.8. For any given input d , the value of C at the end of the i^{th} step of the loop is:

$$C = (d^y \cdot r^z \cdot d^e \cdot r^{(e_i + s_i)}) \bmod M \quad (5.7)$$

Where $y = \sum_{j=i+1}^{n-1} e_j 2^{j-i}$ and $z = \sum_{j=i+1}^{n-1} e_j 2^{j-i} + \sum_{j=i+1}^{n-1} s_j 2^{j-i}$

Thus the values of C during the $(i+1)^{th}$ step have four possible cases:

If $e_i = 0$ and $s_i = 0$, then

$$C = (d^{2y} \cdot r^{2z}) \bmod M \quad (5.8)$$

If $e_i = 0$ and $s_i = 1$, then

$$C = (d^{2y} \cdot r^{2z} \cdot r) \bmod M \quad (5.9)$$

If $e_i = 1$ and $s_i = 0$, then

$$C = (d^{2y} \cdot r^{2z} \cdot d \cdot r) \bmod M \quad (5.10)$$

If $e_i = 1$ and $s_i = 1$, then

$$C = (d^{2y} \cdot r^{2z} \cdot d \cdot r^2) \bmod M \quad (5.11)$$

Thus from the above equations, the power consumed during the calculation of (5.8) will be significantly different from the other three cases. As with ECC, this will be the case when an iterative procedure for the calculation of modular exponentiation, such as one based on Montgomery's modular multiplication algorithm, is used. This possible power attack to Kim's countermeasure will be described in more detail in subsequent sections.

5.3.4 Proposed Countermeasure against DPA for RSA

Kim's countermeasure has the potential to leak sensitive information due to the constant initial value, $T[00]=1$ in Algorithm 5.8. Instead of using a constant initial value, we propose to assign a random value, $T[00]=r^{\phi(M)}$, thus avoiding the possible attack to Kim's countermeasure. Algorithm 5.9 shows our modified countermeasure against DPA.

Algorithm 5.9 Our proposed method to compute d^e for RSA

Input: d, e

Output: $C = d^e \bmod M$

Pre-computation

$s = \phi(M) - e$, where $\phi(M)$ is Euler phi-function

Choose a random number r ,

$T[00] = r^{\phi(M)}$, $T[01] = r$, $T[10] = d \cdot r$, $T[11] = d \cdot r^2$

1. $C = T[00]$
2. **For** $i = n-1$ **Downto** 0
 - $C = C^2 \bmod M$
 - $C = C \cdot T[e_i s_i] \bmod M$

EndFor

3. **Return**($C = d^e$)
-

The final value of Algorithm 5.9 is $d^e = (d^e \cdot r^{2x\phi(M)}) \bmod M$, where x is the number of times that $e_i s_i = 00$. Based on the characteristic of the Euler-phi function [Piep03], $r^{\phi(M)} \bmod M = 1$, and hence $r^{x\phi(M)} \bmod M = 1$. Thus:

$$C = (d^e \cdot r^{2x\phi(M)}) \bmod M = d^e \bmod M \quad (5.12)$$

Thus the calculation of C in (5.8) when $e_i s_i = 00$ becomes $C = (d^{2 \cdot y} \cdot r^{2 \cdot z} \cdot r^{2x\phi(M)}) \bmod M$ with a power consumption similar to (5.9), (5.10), and (5.11). Then, even if the attacker knows the highest bits $e_{n-1}, \dots, e_{i+1}$ of the secret e , the power consumption during the $(i+1)^{th}$ step when computing C will not be significantly different, even when $e_i = 0$ and $s_i = 0$. Our proposed modified countermeasure avoids the weak point of Kim's countermeasures without increasing the computational complexity. It also has the advantage of avoiding the computation of the inverse modular exponentiation of the random number r . Compared with Kim's algorithm, our algorithm is a more efficient method to defend against SPA and DPA and provides a cost-effective method for computing modular exponentiation.

5.4 Evaluation of the Proposed Countermeasures

In this section, the performance and computation cost of our proposed algorithm is discussed. The computational cost of the proposed countermeasure for ECC is calculated. Algorithm 5.10 gives the generalized form for computing the product of e and P for ECC where ECADD and ECDBL represent point adding and point doubling functions, respectively.

Algorithm 5.10 Our proposed method to compute eP for ECC in a general computation form

Input: e, P

Output: $Q = eP$

Pre-computation

$s = \# \mathcal{E} - e$, where $\# \mathcal{E}$ is the number of points of the curves

Choose a random elliptic point R

$T[00] = \# \mathcal{E} R, T[01] = R, T[10] = ECADD(P, R),$

$T[11] = ECADD(P, ECDBL(R))$

1. $Q = T[00]$

2. **For** $i = n-1$ **Downto** 0

$Q = ECDBL(Q)$

$Q = ECADD(Q, T[e_i s_i])$

EndFor

3. **Return** ($Q = eP$)

Appendix A describes a number of different implementations for ECC addition and doubling operations. If M , S and I represent the computation cost of the multiplication, square and inverse operations on the finite field, the costs for EC addition ($ECADD$) and EC doubling ($ECDBL$) operations for the various forms can be estimated as in Table 5.1. Table 5.2 shows the computational cost of M , S and I in clock cycles, where n is the key length. Table 5.3 shows different implementation costs for an elliptic curve calculation, including $ECADD$ and $ECDBL$, for a typical case with 160-bit width and 5 points. As the M , S and I values represent the major cost associated with the countermeasure, the pre-computation of $T[00]$, $T[01]$, $T[10]$ and $T[11]$ have been ignored.

Table 5.1 Computation cost of $ECADD$ and $ECDBL$ for different EC forms

Forms	Operation	Computation Cost
Simplified Weierstraß Form[Piep03]	$ECADD_{we}(P, Q)$	$2M + S + I$
	$ECDBL_{we}(P)$	$2M + 2S + I$
Homogeneous Projective Coordinate [Chud86, Cohe98]	$ECADD_{ho}(P, Q)$	$12M + 2S$
	$ECDBL_{ho}(P)$	$7M + 5S$
Jacobian Projective Coordinate[Piep03]	$ECADD_{ja}(P, Q)$	$12M + 4S$
	$ECDBL_{ja}(P)$	$4M + 6S$
Montgomery Form[Izu03, Fisc02]	$ECADD_{mo}(P, Q)$	$8M + 3S$
	$ECDBL_{mo}(P)$	$3M + 4S$
Hessian Form[Joye01, Smar01]	$ECADD_{he}(P, Q)$	$12M + 6S$
	$ECDBL_{he}(P)$	$9M + 3S$

Table 5.2 Computation cost of M , S and I

	Computation Cost (Clock Cycles)
M	$n+1$
S	$n+1$
I	$(n/2 + 1) \sim n+1$

Table 5.3 The computational cost of a 160-bit, 5 point elliptic curve computation

Forms	Total Computation Cost (Clock Cycles)
Simplified Weierstraß Form[Piep03]	$(1280n + 1440) \sim (1440n + 1440)$
Homogeneous Projective Coordinate[Chud86, Cohe98]	$4160n + 4160$
Jacobian Projective Coordinate[Piep03]	$3160n + 3160$
Montgomery Form[Izu03, Fisc02]	$2880n + 2880$
Hessian Form [Joye01, Smar01]	$4800n + 4800$

The computation cost of the EC calculation using Montgomery's implementation is significantly lower than the other implementations without division operations. Compared to the reported computation cost for Mamiya's implementation of the extended binary-based method with random initial point (EBRIP) with a bit width of 160 and 5 points ($2780n + 2780$) [Mami04], our proposed method using

Montgomery's implementation achieves an almost identical performance. Moreover, our computation cost is almost the same as for Kim's implementation, with both avoiding the inverse modular exponentiation computation of the random number r and the subtraction of the random elliptic point R (see algorithm 5.6). Our proposed countermeasure avoids an initial value power analysis attack by randomizing the initial value $T[00]$.

For RSA the major cost is in modular exponentiation, In the proposed implementation of Algorithm 5.9, the additional computational cost is in computing $T[00] = r^{\phi(M)}$. However as this is pre-computed on entry to the algorithm, it does not represent a significant cost. Thus, while the computation cost of our proposed algorithm for RSA is slightly higher than that of Kim's, due to the different initial values, our algorithm performs better under an "initial value DPA" attack scenario.

Unfortunately, much of the literature lacks any experiment results for differential power analysis. To perform a power analysis, we use Active HDL, Synopsys Design Compile, Synopsys VCS, Synopsys Prime Power and the Debussy tool chain to simulate the power analysis of Kim's algorithm and our proposed algorithm. We present a power analysis (Figure 5.1) showing part of the exponentiation calculation for RSA. To keep the analysis manageable, we use a 32 bit key length with the initial values of $T[00]$, $T[01]$, $T[10]$ and $T[11]$ pre-computed. The following parameters (expressed in hexadecimal) are used in the simulation:

1. The two prime numbers are p equals $h'935$ and q equals $h'9F7$
2. The modulo number N equals $h'5BBF23$ and $\phi = (p-1)(q-1) = h'5BABF8$
3. The secret key e equals $h'38131F$
4. The message m equals $h'A500F11A$
5. The random number r equals $h'1058ED3C$.

We use Montgomery modular multiplication in the modular exponentiation calculation for computing RSA. There are two steps for computing the modular multiplication using Montgomery's algorithm. In the first step we compute $C = A \cdot B \cdot R^{-1} \bmod M$, and in the second step we compute $S = C \cdot R^2 \cdot R^{-1} \bmod M$. Thus there are four steps in the calculation of modular exponentiation associated with two lines inside the **For** loop of Algorithm 5.9. These four steps are labeled in Figure 5.1 as *square1*, *square2*, *multiply1* and *multiply2*, respectively.

During the two multiply steps of Figure 5.1 *shift_a* is $T[e_i s_i]$ in the first step and R^2 for the second step, while $e_i s_i$ represents the value of $e_i s_i$. Therefore, during the

computation of Kim's algorithm (Figure 5.1 (a)), when $e_i s_i = 00$ in the first multiplication step, $T[00]$ (*shift_a*) will start at 1 and after the first clock cycle will decrement and stay at 0 for the remaining $n/2-1$ cycles. In Figure 5.1 (b), when $e_i s_i = 00$, $T[00]$ (*shift_a*) will start at the value $h'047F145$ ($r^{\phi(M)}$) and will only reach 0 at the end of the first multiplication step.

The advantage of our proposed algorithm can be seen by comparing the power simulation results, shown in Figure 5.2, for Kim's algorithm and our proposed algorithm. The dashed box from 6100 ns to 12900 ns represents the equivalent period shown in Figure 5.1. Note that the simulation times in Figure 5.1 and Figure 5.2 have been normalized to eliminate a 6ns gate level timing delay inherent in the power simulation. Significant power variations (spikes) can be seen in Kim's algorithm when $e_i s_i = 00$. Closer inspection reveals that they coincide with the transitions between the various multiply and square steps. The power variation is noticeably higher at the transition from the *multiply1* step to the *multiply2* step (at 7200 ns, 9800 ns and 12400 ns), while it is almost absent for the transition from *square2* to *multiply1* (at 6600 ns, 9200 ns and 11800 ns). This power variation at the transition between calculation steps is due to the sudden change in the transistors' internal state at the start of the calculation from the 0's which had propagated through the architecture due to the value of *shift_a* ($= 0$) at the end of the previous calculation step. This then provides a mechanism to identify when $e_i s_i = 00$, and hence provides a possible avenue for attack. The power consumption of the proposed algorithm exhibits much less variability, particularly at the critical points when $e_i s_i = 00$. This is because, as can be clearly seen at the start of the *multiply1* step in Figure 5.1, *shift_a* is non-zero for the duration of the calculation as $T[00]$ has the initial value $h'047F145$ instead of 1. Due to the additional processing when $e_i s_i = 00$, the proposed algorithm consumes slightly more power (1.381 mW compared to Kim's 1.359 mW). While the above simulations were performed using a 32 bit key length, the power waveform for a 2048 bit key length will exhibit similar characteristics.

Figure 5.1 The computation wave form of (a) Kim's algorithm and (b) the proposed algorithm

Figure 5.2 The power consumption waveform of (a) Kim's algorithm and (b) the proposed algorithm

5.5 Conclusion

In this chapter, Mamiya's [Mami04] countermeasure was introduced as an efficient method for resisting against SPA, DPA, RPA and ZPA attacks. Unfortunately, this method is only suitable for ECC. Kim proposed countermeasures against power analysis attacks in which he extended Mamiya's method to RSA. Kim's algorithm also avoided the need for calculating the inverse modular exponentiation of the random number r . However, the initial value of $T[00]=1$ for RSA could result in the leakage of sensitive information when $e_i = 0$ and $s_i = 0$. A power attack scenario for the initial value of Kim's algorithm is established. We then proposed an efficient scheme against power analysis attacks of modular exponentiation for ECC and RSA without significantly increasing the computation complexity. Finally, power simulation of the proposed modular exponentiation for RSA shows that our proposed method is more robust towards power analysis attacks due to a notable reduction in the power variations. As modular multiplication is also the critical computation for ECC, we would expect a similar result for an ECC power simulation.

Chapter 6 A DPA-resistant Unified Architecture for Public Key Cryptosystems

6.1 Introduction

Signcryption is a cryptographic primitive supporting both confidentiality and authentication. A signature scheme and an encryption scheme provide authentication and confidentiality respectively. In this chapter, we present the framework for a DPA-resistant unified cryptographic coprocessor for signing, encryption and signcryption, which has high performance and area-efficiency. The coprocessor is designed to resist probing attacks on the internal bus and memory. Private keys are maintained within the execution unit to reduce the possibility of leakage during cryptographic operations. The unified execution unit includes scalar modular multiplication, a random number generator, a prime random number generator and enveloping units. The architecture makes use of the signed digit (SD) adder based unified multiplier over $GF(p)$ and $GF(2^p)$ discussed in Chapter 4, and the DPA-resistant algorithm for RSA and ECC operations on $GF(p)$ and $GF(2^p)$ discussed in Chapter 5 of this thesis.

6.2 Unified Architecture for Signcryption

In this section, a unified framework for signcryption is proposed consisting of a signature scheme, an encryption scheme, and a signcryption scheme. The details of the three schemes can be found in Appendix B. We divide the security application into three levels: the software level, the firmware level and the hardware level, as shown in Figure 6.1. The cryptographic protocols are at the firmware level which is further subdivided into the logic and operations parts. The signing, encryption, and signcryption processes are in the logic part while the key generation and modular exponentiation processes are performed in the operations part. The hardware level contains the structures and control to perform the low level cryptographic operations, such as modular multiplication, modular addition, modular subtraction, etc.

6.3 DPA-resistant Modular Exponentiation

RSA and ECC are two commonly used public key cryptography techniques for providing secure and authenticated communications. However, power analysis attacks, such as SPA and DPA, have recently emerged as efficient methods for attacking the physical implementations of cryptographic algorithms [Koch99]. These attacks target the calculation of modular exponentiation using a secret key exponent, by observing

Table 6.1 Procedures for signature, encryption and signcryption schemes

Signature Scheme (S)	<ul style="list-style-type: none"> ▪ $S\text{Gen}$ generates two pairs of keys $\{SDK_s, VEK_s\}$ & $\{SDK_r, VEK_r\}$ $pub = \{VEK_s, VEK_r\}$ ▪ Signing Key $SK = \{SDK_s, pub\}$ ▪ Verification Key $VK = \{SDK_r, pub\}$ ▪ Signing: $c = Sig(m)$ ▪ Verification: $m = Ver(c)$
Encryption Scheme (E)	<ul style="list-style-type: none"> ▪ $E\text{Gen}$ generates two pairs of keys $\{SDK_s, VEK_s\}$ & $\{SDK_r, VEK_r\}$ $pub = \{VEK_s, VEK_r\}$ ▪ Encryption Key $SK = \{SDK_s, pub\}$ ▪ Decryption Key $VK = \{SDK_r, pub\}$ ▪ Encryption: $c = Enc(m)$ ▪ Decryption: $m = Dec(c)$
Signcryption Scheme (SE)	<ul style="list-style-type: none"> ▪ $SE\text{Gen}$ generates two pairs of keys (SDK_s, VEK_s) & (SDK_r, VEK_r) $pub = \{VEK_s, VEK_r\}$ ▪ Signing Encryption: $c = SigEnc_{\{SDK_s, VEK_s\}}(m)$ ▪ Verification Decryption: $m = VerDec_{\{SDK_r, VEK_r\}}(c)$

Figure 6.1 The overall architecture

that the power consumption is reduced when the secret key has a zero bit. In Chapter 5, a modified countermeasure against DPA is proposed for RSA cryptosystems. The DPA-resistant algorithm for computing eP is given in Algorithm 5.5, and for computing d^e is given in Algorithm 5.9 of Chapter 5. The proposed crypto-processor makes use of these algorithms when computing modular exponentiation. Since modular exponentiation is the main operation in RSA and ECC computations, we use this as an example to illustrate the effectiveness of our proposed architecture.

6.4 The Proposed Crypto-processor

A crypto-processor is defined as a general-purpose cryptographic computing machine that is able to withstand physical attacks and side channel attacks. Physical and side channel attacks are a powerful method to recover sensitive data from smart cards or other physical devices by using timing, power and fault analysis, etc. We aim to design a crypto-processor which exhibits high performance and high security, and thus could provide a secure platform for specific applications such as RSA, ElGamal and Diffie-Hellman key agreement. As the target platform for our design is an ASIC, some attacks, such as temperature imprinting, voltage imprinting, high and low voltage and probing attacks, that rely on environmental changes can be dealt with by adding sensors to the design. Other attacks relating to information leakage, such as by the internal bus, the memory, power and timing, can be resisted by our design. Figure 6.2 shows the structure of the proposed crypto-processor, while Figure 6.3 shows the structure of the unified execution unit which includes the random number generator, the large prime random number generator, the enveloping unit and the modular multiplication unit. The proposed crypto-processor can support two modes: a normal mode and a secure mode. Modular multiplication and modular exponentiation are performed in normal mode, while private key operations and the protection of sensitive data during transfers from memory to the execution unit are done in secure mode. In the proposed architecture, sensitive data is transferred via the internal bus. As the data could be detected by probing attacks, it is encrypted before being sent to the memory via the internal bus. Sensitive data can also be determined by scanning the contents of memory. One simple way to tackle the leakage of sensitive data from memory is to reorder the data sequences stored in memory, as was done in the design presented in Chapter 4. However, in the framework presented here, we have elected to store encrypted sensitive data in memory, which makes it more difficult to attack even when the memory sequence order is known. A number of other changes to the preliminary design presented in Chapter 4 (for reference see Figure 4.2) have been made to improve the security of the proposed crypto-processor. These include: moving the memory from the common bus interface to the internal bus so that private and session

keys are not externally accessible; encrypting the data stored in memory; and forcibly overwriting the keys, using a fixed value, upon key expiry.

Figure 6.2 The proposed architecture for a crypto-processor

Figure 6.3 The unified execution unit for computing over $GF(p)$ and $GF(2^p)$

Figure 6.4 The format of the crypto-processor instructions

Data and micro-instructions are sent to the execution unit and the instruction memory via a conventional computer interface as shown in Figure 6.2. The 16-bit micro-instructions for the proposed crypto-processor can have one of three formats as shown in Figure 6.4. The first two bits determine the format, while the next three bits represents the key length. The subsequent three bits represent the radix value, while the remaining 8 bits represent the operation code (op-code). The definition of key length and radix value is show in Table 6.2.

Table 6.2 The codes for the key length and radix value

Key Length	Code	Radix Value	Code
128	001	1	001
160	010	2	010
256	011	4	011
512	100	8	100
768	101	16	101
1024	110	32	110
2048	111	NA	NA

6.4.1 Unified Execution Unit

The unified execution unit, capable of performing cryptographic calculations over $GF(p)$ and $GF(2^p)$, is the main computation engine of the proposed crypto-processor. Figure 6.3 shows the structure of the proposed unified execution unit which consists of a random number generator, a large prime random number generator, the modular multiplication logic and an enveloping unit. Public and private keys, for setting up RSA, are generated by the large prime random number generator. As modular exponentiation is needed for testing the primality of the generated random numbers, the modular multiplication unit can be reused for key generation. The possibility of the leakage of sensitive data is decreased by the computation and transformation of private keys within the unified execution unit. The enveloping unit encrypts and decrypts the private keys whenever they are transferred between the execution unit and memory via the internal bus. The contents of memory are the encrypted sensitive data.

The unified execution unit, supporting modular multiplication over both $GF(p)$ and $GF(2^p)$ fields using the same arithmetic unit, is essentially identical to the execution unit described in Chapter 4 (Figure 4.2), with the addition of the random number generator, the large prime random number generator and the enveloping unit. The data converter, between the unified execution unit and the bus interface of Figure 6.2, is needed to convert the SD value back into a normal binary representation, as described

in Chapter 4 (see Algorithm 4.2). This unit has been implemented in VHDL and has been successfully synthesized to FPGA and 0.09 μm ASIC.

6.4.2 Modular Multiplication

The Main challenge for the effective implementation of the execution unit is the computational cost of modular multiplication, as the speed of modular multiplication is critical to the performance of modular exponentiation, which is a key calculation in many cryptographic algorithms. Thus, it is not surprising that the proposed crypto-processor's critical path is the modular multiplication unit.

Algorithms for the calculation of $A \cdot B \bmod M$ were presented in Chapter 3 and Chapter 4, with an efficient implementation of a unified modular multiplication unit using a signed-digit adder structure for computation in both $GF(p)$ and $GF(2^p)$ being described in Chapter 4. The algorithms describing the implementation of modular multiplication using a signed-digit adder structure given in Chapter 4 are repeated here, for convenience, as Algorithm 6.1 and Algorithm 6.2. It should be noted that the output of Algorithm 6.2 is $S_n = A \cdot B \cdot R^{-1} \bmod M$ and so two separate steps are needed to calculate the modular multiplication as:

$$A \cdot B \bmod M = (A \cdot B \cdot R^{-1} \bmod M) \cdot R^2 \cdot R^{-1} \bmod M \quad (6.1)$$

Similarly, $a(x) \cdot b(x) \bmod m(x)$ can be obtained as:

$$a(x) \cdot b(x) \bmod m(x) = (a(x) \cdot b(x) \cdot x^{-n} \bmod M(x)) \cdot x^{2n} \cdot x^{-n} \bmod m(x) \quad (6.2)$$

The SXOR() operation in Algorithm 6.1 and the SADD() operation in Algorithm 6.2 represent XOR and addition operations in the signed-digit number system. The full details can be found in chapter 4.

The modular multiplication is performed in the unified execution unit using a sequence of micro-instructions. Firstly, the data must be loaded into internal registers in the execution unit. A modular multiplication micro-instruction is issued, and lastly the result must be transferred back to memory. Figure 6.5 shows the structure of the scalable modular multiplication operation. B and M are sent to the modular multiplication unit ($MODM$) via the B_Bus and M_Bus, respectively. The state machine controller configures the $MODM$ unit according to the different radix values (k) and the different key lengths (n).

Figure 6.5 The structure of $MODM(A, B, M, k, n)$

Algorithm 6.1 the proposed algorithm to compute $s(x) = a(x) \cdot b(x) \cdot x^{-n} \bmod m(x)$ based on a signed-digit adder

Input: $a(x)$, $b(x)$ and $m(x)$

Output: $s(x) = a(x) \cdot b(x) \cdot x^{-n} \bmod m(x)$

$$a(x) = \sum_{i=0}^{n-1} a_i x^i, \quad a_i \in GF(2),$$

$$b(x) = \sum_{i=0}^{n-1} b_i x^i, \quad b_i \in GF(2),$$

$$m(x) = x^n + \sum_{i=0}^{n-1} m_i x^i, \quad m_i \in GF(2)$$

-
1. $s(x)_0 = 0$
 2. **For** $i = 0$ **To** $n - 1$
 - $s(x)_i = SXOR(s(x)_i, a_i b(x))$
 - $q_i = s_0(x)_i$
 - $s(x)_i = SXOR(s(x)_i, q_i m(x))$
 - $s(x)_{i+1} = rightshift(s(x)_i)$
 - EndFor**
-

Algorithm 6.2 the proposed algorithm to compute $A \cdot B \cdot R^{-1} \bmod M$ based on a signed-digit adder

Input: A, B and M

Output: $A \cdot B \cdot R^{-1} \bmod M$

$$M = \sum_{i=1}^{n-1} 2^i m_i, m_i \in \{0,1\},$$

$$B = \sum_{i=1}^{n-1} 2^i b_i, b_i \in \{0,1\},$$

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

$$A, B < M, M < R = 2^n, M' = -M \bmod 2 = M, \gcd(2, M) = 1.$$

1. $S_0 = 0$
 2. **For** $i = 0$ **To** $n - 1$
 - $\tilde{S}_i = SADD(\tilde{S}_i, a_i B)$
 - $q_i = \tilde{S}_{i_0}$
 - $\tilde{S}_i = SADD(\tilde{S}_i, q_i M)$
 - $\tilde{S}_{i+1} = \text{rightshift}(\tilde{S}_i)$
 - EndFor**
 3. $S_n = \text{convert}(\tilde{S}_n)$
-

6.4.3 Pseudo Prime Number Generator

Setting up the RSA keys is the first step of an RSA public key cryptosystem. This involves choosing two large prime numbers. Due to the difficulty in generating large prime numbers, cryptography algorithms typically use pseudo-prime numbers, i.e. numbers that appear to be prime but may not actually be prime. The Miller-Rabin probabilistic primality test is a well recognized technique for producing good quality RSA keys [Rabi80]. Thus the numbers generated by the random number generator and the large prime number generator need to pass the Miller-Rabin test. The generation of a pseudo prime number a involves checking that it satisfies the $2s$ equations given in Table 6.3 below. Where s must satisfy $n-1 = 2^{sr}$, with r being an odd number and n being the modulo number selected for testing. As can be seen from the above procedure, the main operation in the generation of a large prime number is modular exponentiation.

Table 6.3 The equations for determining primality

Step1	$a^r \bmod n \neq 1$
Step2	$a^r \bmod n \neq -1$
Step3	$a^{2^r} \bmod n \neq 1$
Step4	$a^{2^r} \bmod n \neq -1$
Step5	$a^{4^r} \bmod n \neq 1$
Step6	$a^{4^r} \bmod n \neq -1$
...	...
Step $2s-1$	$a^{2^{s-1}r} \bmod n \neq 1$
Step $2s$	$a^{2^{s-1}r} \bmod n \neq -1$

6.4.4 The Enveloping Unit

Unencrypted data is susceptible to probing attacks to the internal chip buses. A Non-linear feedback shift register (NLFSR) [Mene97] can be used to encrypt sensitive data to prevent probing attacks. This can be implemented by using a *clock-controlled generator* to introduce nonlinearity into a linear feedback shift register (LFSR) based key stream generator by having the output of one LFSR control the *clocking* (i.e., stepping) of a second LFSR. The first clock controlled generator, called an *altering generator*, uses a LFSR, R_1 , to control the stepping of two other LFSRs, R_2 and R_3 [Mene97]. The key stream produced is the XOR of the output sequences of R_2 and R_3 . We use the following LFSRs:

$$R_1 = \langle 128, 1 + D^2 + D^{27} + D^{29} + D^{128} \rangle$$

$$R_2 = \langle 131, 1 + D + D^{47} + D^{48} + D^{131} \rangle$$

$$R_3 = \langle 133, 1 + D + D^{51} + D^{52} + D^{133} \rangle$$

The lifetime of our altering generator is $2^{128} (2^{131} - 1)(2^{133} - 1)$, which is also the life time of the set of private keys.

Figure 6.6 The process of encryption and decryption of private keys for transfer between the execution unit and memory

The encryption and decryption of private keys for transfer between the execution unit and memory is shown in Figure 6.6. The keys are generated by the random number generator and the large prime number generator. Some long-term private keys need to be stored to memory and must be transferred across the internal buses. Private keys are encrypted before being stored in memory. Prior to transferring the private key k to the internal bus, it will be encrypted into k' by the enveloping unit. The encrypted key k' is then stored in memory. When the private key k is required, k' will be transferred from memory and decrypted into k prior to use in the execution unit.

The long term key k_{ENV} , the session key k_s and the private key k are all generated within the execution unit. The private key k is encrypted using session key k_s and the session key k_s is encrypted using long term key k_{ENV} when the private key is sent out to memory. Each session k_s is generated for a single storage operation and is deleted after a single use. When the execution unit receives a private key from memory, the decryption procedure is simply the reverse of the encryption process. The format of the data stored in memory is $(E_{k_s}(k, RND), E_{k_{ENV}}(k_s, RND))$, making it difficult to decrypt the private key even when the memory contents are known. The definition and life time of these keys are shown in Table 6.4:

Table 6.4 Definition and life time of different keys

	Life Time	Definition	Status
k	Long term key	Private key	Existing when power on and off
k_{ENV}	Long term key	Envelop key	Existing when power on and off
k_s	short term key	Session key	Delete the old session key & generate a new one when the power is turned on

6.5 RSA Calculations

In this subsection, we introduce the procedure for setting up the RSA private keys, and for computing the modular exponentiation, as these represent the two most common cryptographic operations. The proposed crypto-processor has 16-bit instructions and a 32-bit data path. There are three types of instructions: data transfer, execution and control instructions. In the proposed design, each instruction is decoded into a series of states for controlling the operation of the crypto-processor.

As a private key has a long life time, the challenge for the proposed design is in the selection and storage of those keys. An RSA private key k is co-prime with $(p-1)(q-1)$. Euclid's method [Knut81] is used for computing $\gcd(k, \phi)$, where $\phi = (p-1)(q-1)$ and p

and q are two large primes. The procedure for setting up the RSA private keys is described in the pseudocode [Tsoi03] in Algorithm 6.3. It can be seen that to produce an RSA private key, it is necessary to test the primality between the random number k and ϕ . The specific random number k will be output as the private key.

Algorithm 6.3 Produce a private key

Input : X, Y , two large primes p and q

Output: k

1. $X = \phi$; $\phi = (p-1)(q-1)$;
2. **get_random:**
 $Y = k$;
3. **gdc_sub:**
 - 3.1 $Y = Y - X$;
 - 3.2 IF ($Y == 1$) RETURN ($k=Y$); ENDIF
 - 3.3 IF ($Y == 0$) GOTO **get_random** k ; ENDIF
 - 3.4 IF ($Y < 0$) $Y = Y + X$; ENDIF
4. **swap** X, Y ;
5. GOTO **gdc_sub**;

Figure 6.7 The flowchart (a) for setting up the RSA keys and (b) for performing modular exponentiation

Public and private keys need to be generated within the execution unit. The first step, when setting up an RSA-based communication, is the generation of the 1024-bit public key N and the 1024-bit private key k using two 512-bit prime numbers p and q as described above. The firmware level process for performing key generation, issues a sequence of instructions which implement the flow shown in Figure 6.7(a).

The second step is the encryption which needs to compute $A^k \bmod M$, where A and M are of 1024-bit length. The firmware level process for performing modular exponentiation issues a sequence of instructions which implement the flow shown in Figure 6.7(b), which is derived from Algorithm 5.9 of Chapter 5.

6.6 Conclusion

Security is becoming a more and more important issue in embedded systems. Tamper attacks and physical attacks attempt to exploit the weak design features of some hardware based cryptographic algorithms and processors. Our proposed crypto-processor framework has the ability to resist probing attacks and power attacks using bus encryption and a unified execution unit. The unified execution unit can support the operations over $GF(p)$ and $GF(2^p)$, and thus can be applied to RSA and ECC crypto operations. The execution unit has built in countermeasures which give it the ability to better resist against side channel attacks, especially DPA attacks.

Chapter 7 Conclusions and Future Work

7.1 Conclusions

The work presented in this thesis is aimed at exploring a unified public key cryptographic processor with the ability to defend against power analysis attacks. The main emphasis was placed on a unified arithmetic unit for supporting addition, subtraction, multiplication and inversion over $GF(p)$ and $GF(2^p)$. A high performance modular multiplication arithmetic unit has been proposed to shorten the total computation time. An “initial value DPA” attack is presented as an efficient method to break some of the countermeasures introduced to counter power analysis attacks in the literature. We also propose a countermeasure against this kind of attack. We then extend our work to develop a framework for a cryptographic coprocessor, based on the results of our earlier work, with the aim of protecting the sensitive information from cryptanalysis. The proposed coprocessor has an independent instruction set, a unified arithmetic unit, an enveloping unit, a secure internal bus and encrypted memory.

The contributions, outlined in detail in earlier chapters, can be summarized as follows:

7.1.1 High Performance Modular Multiplication

We have proposed a modified Montgomery modular multiplication algorithm to reduce the total computation time to nearly half that presented in [Blum01]. When used for calculating modular exponentiation, the implementation results show that our proposed algorithm is faster and consumes less power compared with existing methods from the literature, when implemented on either FPGA or ASIC platforms.

7.1.2 Unified Modular Arithmetic Unit over $GF(p)$ and $GF(2^p)$

A novel signed digit adder has been employed to realize a new unified modular arithmetic unit over $GF(p)$ and $GF(2^p)$. Due to the characteristics of the problem, signed digit arithmetic was used to provide a balance between the carry propagation modes of operation and the carry-less modes. Compared to a conventional full adder implementation, our signed digit adder has only two levels of LUT in the critical path and is not constrained by the need for fast carry propagation. Our proposed adder is used as the fundamental module for realizing a unified modular arithmetic unit over $GF(p)$ and $GF(2^p)$. A unified modular arithmetic unit, including dual-field modular multiplication, dual-field modular addition/subtraction and dual-field modular inversion was then developed. Simulation results show that our proposed architecture is faster and uses less hardware area than existing methods from the literature.

7.1.3 Countermeasures against Power Attacks

Side channel attacks are an efficient way to deduce the sensitive information from the hardware implementation of security applications. Of these attacks, power analysis attacks are particularly effective. Two methods exist to deal with these kinds of attacks. One is to modify the algorithms by making the outputs independent from any inputs. The other is to modify the hardware implementation by adding operations to maintain constant power consumption. We have addressed one possible attack, which we term the “initial value DPA” attack, and have shown that it is possible to use it to deduce sensitive data. In order to defend against this kind of attack, we have proposed a refined countermeasure using a variable value instead of the constant initial value currently used in implementations from the literature. Simulation results of the power-timing waveform, when calculating modular exponentiation for RSA, clearly show that our proposed method has the ability to defend against the “initial value DPA” attack. The proposed countermeasure can also be extended to ECC.

7.1.4 A DPA-resistant Unified Cryptographic Coprocessor

Side channel attacks, and specifically probing attacks, can access information from the internal bus and memory. Thus, sensitive key data should be protected when transferred through these units. We proposed a framework for a DPA-resistant unified cryptographic coprocessor which has the ability to protect information, such as the private keys, when transferred across the internal bus or when stored in memory. This framework makes use of our improved countermeasures against “initial value DPA”, SPA, DPA, ZPA and RPA attacks. The proposed architecture includes a unified execution unit, a controller, signed digit data converter and memory. The unified execution unit is based on our unified modular arithmetic unit and includes a random number generator, a pseudo-random number generator and an enveloping unit.

7.2 Future Work

Based on the work presented in this thesis, we suggest several interesting and relevant topics for further research.

7.2.1 Supporting Private Key Cryptosystems

As well as public key cryptosystems, there exist a number of private key cryptosystems. A private key cryptosystem is a cryptographic method in which the same key is used to encrypt and decrypt the message. Private-key algorithms include the Data Encryption Standard (DES), triple-DES (3DES), the Advanced Encryption Standard (AES, also known as Rijndael), Blowfish, Twofish RC2, RC4, RC5 and RC6. Private key cryptosystems can be integrated into our proposed architecture in the future to support a wider range of security applications.

7.2.2 Application Specific Instruction-set Processor

An Application Specific Instruction-set Processor (ASIP) is a processor designed for a particular application or a set of applications. Another goal of our future research work is to design a unified security coprocessor with application specific instructions. We could then exploit special characteristics of the security application to meet the desired performance, power and cost requirements. This would also be advantageous for implementing security applications whose standards are subject to minor revision, as ASIC implementations are difficult to change at a later stage. In such a situation, an ASIP could offer the required flexibility and performance at a lower overall cost than a General Purpose Processor (GPP) with an attached crypto-coprocessor.

7.2.3 Defending against Side Channel Attacks (Timing and Fault Attacks)

Apart from power attacks, there are a number of other side channel attacks, such as timing and fault attacks, which pose a significant threat to hardware implementations. A timing attack is based on measuring the time it takes for a unit to perform certain operations. This information can lead to information about the secret keys. For example, by carefully measuring the amount of time required to perform private key operations, an attacker might find the fixed Diffie-Hellman exponents or factor RSA keys, etc. Timing attacks can be countered by engineering cryptographic hardware with uniform execution time properties. Fault analysis relates to the ability to investigate a cipher and extract keys, by examining operation during natural faults or by generating faults, in a system that is in the possession of the attacker. Faults are most often caused by changing the voltage, tampering with the clock, or by applying radiation of various types. To prevent fault analysis attacks, hardware must be designed to perform uniformly or enter a secure and consistent state under abnormal conditions. Defending against side channel attacks, especially timing and fault attacks, will be one of the more challenging avenues for research in this area in the future.

7.2.4 Develop a Prototype of the Proposed Architecture

In this thesis, we have developed a framework for a secure crypto-processor. This could be extended to the prototype stage. Alternatively, the various hardware modules developed could be refined and deployed as IP cores for secure embedded applications.

Bibliography

- [Agne90] G. B. Agnew, R. C. Mullin, and S. A. Vanstone. *A fast elliptic curve cryptosystem*. Advances in Cryptology - EUROCRYPT'89, LNCS 434, Springer-Verlag, pages 706-708, 1990.
- [Agne93] G. B. Agnew, R. C. Mullin, and S. A. Vanstone. *An implementation of elliptic curve cryptosystem over $F_{2^{155}}$* . IEEE J. Selected Areas Communication, volume 11, No. 5, pages 804-813, 1993.
- [Akis03] T. Akishita and T. Takagi. *Zero-Value point attacks on elliptic curve cryptosystem*. ISC 2003, LNCS 2851, Springer-Verlag, pages 218-223, 2003.
- [Alia91] G. Alia and E. Marinelli. *A VLSI modulo m multiplier*. IEEE Transactions on Computers, volume 40, No. 7, pages 873-878, July 1991.
- [Alve04] T. Alves and D. Felton. *TrustZone: integrated hardware and software security*. Technical report, July 2004. ARM Limited.
- [An02] J. H. An, Y. Dodis, and T. Rabin. *On the security of joint signature and encryption*. Advances in Cryptology – EUROCRYPT 2002, LNCS 2332. Springer-Verlag, pages 83–107, 2002.
- [Atru65] A. J. Atrubin. *A one-dimensional real-time interactive multiplier*. IEEE Transactions on Electron. Comput., volume 14, pages 394-399, 1965.
- [Aviz61] A. Avizienis. *Signed-digit representation for fast parallel arithmetic*, IRE Trans. Electron. Computer, volume 10, page 389-400, September 1961.
- [Baja98] J. C. Bajard, L. S. Didier and P. Kornerup. *An RNS Montgomery modular multiplication algorithm*. IEEE Transactions on Computers, volume 47, No. 7, pages 766-776, July 1998.
- [Baja04] J. C. Bajard and L. Imbert, *A full RNS implementation of RSA*, IEEE Transaction on Computers, volume 53, No. 6, pages 769-774, June 2004.
- [BarEl] H. Bar-El. *Introduction to Side Channel Attacks*. White paper, Discretix Technologies, Ltd. URL: www.discretix.com/PDF/Introduction%20to%20Side%20Channel%20Attacks.pdf
- [Bati02] L. Batina, S. B. Örs, B. Preneel and J. Vandewalle. *Hardware architecture for public key cryptography*. Integration, the VLSI journal, volume 34, pages 1-64, 2002.
- [Bati04] L. Batina, and G. B. Muurling, and S. B. Örs. *Flexible hardware design for RSA and Elliptic curve cryptosystems*. CT-RSA 2004, LNCS 2964, Springer-Verlag, pages 250-263, 2004.
- [Blum99] T. Blum and C. Paar. *Montgomery modular exponentiation on reconfigurable hardware*. Proceedings 14th IEEE Symposium on Computer Arithmetic, pages 70-77, 1999.
- [Blum01] T. Blum and C. Paar. *High-radix Montgomery modular exponentiation on reconfigurable hardware*. IEEE Transactions on Computers, volume 50, issue 7, pages 759-764, July 2001.

-
- [Brie02] E. Brier and M. Joye. Weierstra . *Elliptic curve and side-channel attacks*. PKC 2002, LNCS2274, pages 335-345, Springer-Verlag, 2002.
 - [Boer03] B. D. Boer, K. Lemke, and G. Wieke. *A DPA attack against the modular reduction within a CRT implementation of RSA*. Proceedings Cryptographic Hardware and Embedded Systems, LNCS 2523, Springer-Verlag, pages 228-243, 2003.
 - [Chin06] Y. Chingwei, H. En-Feng and C. Kai-Wen, *An 830mW, 586kbps 1024-bit RSA chip design* DATE'06, volume 2, pages 1-6, 2006.
 - [Cho01] K. S. Cho, J. H. Ryu, and J. D. Cho, *High-speed modular multiplication algorithm for RSA cryptosystem*. The 27th Annual Conference of the IEEE Industrial Electronics Society, pages 479-483, 2001.
 - [Chud86] D. V. Chudnovsky and G. V. Chudnovsky. *Sequences of numbers generated by addition in formal groups and new primarily and factorization tests*. Advances in Applied Mathematics volume 7, issue 4, pages 385-434, 1986.
 - [Cila05] A. Cilaro, A. Mazzeo, N. Mazzocca, and L. Romano, *A novel unified architecture for public key cryptography*, Proceedings IEEE Design, Automation and Test in Europe Conference and Exhibition, volume 3, pages 52-57, 2005.
 - [Cohe98] H. Cohen, A. Miyaji, and T. Ono. *Efficient elliptic curve exponentiation using mixed coordinate*. Proceedings Advances in Cryptography – ASIACRYPT'98, LNCS 1514, Springer-Verlag, pages 51-65, 1998.
 - [Coro99] J. S. Coron. *Resistance against differential power analysis for elliptic curve cryptosystems*. Proceedings Cryptographic Hardware and Embedded Systems, LNCS1717, Springer-Verlag, pages 292-302, 1999.
 - [Crow05] F. Crowe, A. Daly and W. Marnane. *A scalable dual mode arithmetic unit for public key cryptosystems*. International Conference on Information Technology: Coding and Computing ITCC'05. 2005
 - [Daly02] A. Daly and W. Marnane, *Efficient architectures for implementing Montgomery modular multiplication and RSA modular exponentiation on reconfigurable logic*, Proceedings 2002 ACM/SIGDA Tenth International Symposium on Field-Programmable Gate Arrays, pages 40-49, 2002.
 - [Diff76] W. Diffie and M. E. Hellman. *New directions in cryptography*. IEEE Transactions on Information Theory, volume 22, pages 644-654, 1976.
 - [Eldr93] S. E. Eldridge and C. D. Walter. *Hardware implementation of Montgomery's modular multiplication algorithm*. IEEE Transactions on Computers, volume 42, No. 6, pages 693-699, 1993.

- [ElGa85a] T. ElGamal, *A public key cryptosystem and a signature scheme based on discrete logarithms*. Advanced in Cryptology: CRYPTO'84, LNCS 196, Springer-Verlag, pages 10-18, 1985.
- [ElGa85b] T. ElGamal, *A public key cryptosystem and a signature scheme based on discrete logarithms*, IEEE Transactions on Information Theory, volume 31, pages 469-472, 1985.
- [Erns03] M. Ernst, M. Jung, F. Madlener, S. Huss, and R. Blümel. *A reconfigurable system on chip implementation for elliptic curve cryptography over $GF(2^n)$* . Proceedings Cryptographic Hardware and Embedded Systems, LNCS 2523, pages 381-399, 2003.
- [Even91] S. Even. *Systolic modular multiplication*. CRYPTO'90, LNCS 537, pages 619-624, 1991.
- [Fisc02] W. Fischer, C. Giraud, E. W. Knudsen, and J. P. Seifert. *Parallel scalar multiplication on general elliptic curves over $GF(p)$ hedged against non-differential side-channel attack*. IACR, Cryptography ePrint Archive, <http://eprint.iacr.org/2002/007.pdf>, 2002.
- [Fouq03] P. A. Fouque and F. Valette. *The doubling attack - why upwards is better than downwards*. Proceedings Cryptographic Hardware and Embedded Systems, LNCS 2779, Springer-Verlag, pages 268-280, 2003.
- [Gao99] L. J. Gao, S. Shrivastava, and G. Sobelman. *Elliptic curve scalar multiplier design using FPGAs*. Proceedings First Cryptographic Hardware and Embedded Systems, Springer-Verlag LNCS1717, pages 257-268, 1999.
- [Good01] J. Goodman and A. P. Chandrakasan. *An energy-efficient reconfigurable public-key cryptography processor*. IEEE Journal of Solid-State Circuit, volume 36, No. 11, pages 1808-1820, 2001.
- [Gord98] D. M. Gordon. *A survey of fast exponentiation methods*. Journal of Algorithms, volume 27, pages 129-146, 1998.
- [Goub03] L. Goubin. *A refined power-analysis attack on elliptic curve cryptosystem*. PKC 2003, LNCS 2567, Springer-Verlag, pages 199-211, 2003.
- [GroB01] J. GroBschadl. *A bit-serial multiplier architecture for finite fields $GF(p)$ and $GF(2^k)$* . Proceedings Cryptographic Hardware and Embedded Systems, LNCS 2162 Springer-Verlag, pages 202-219, 2001.
- [Gutu03] A. A. Gutub, and A. F. Tenca, and E. Savas, and C. K. Koc. *Scalable and unified hardware to compute Montgomery inverse in $GF(p)$ and $GF(2^n)$* . Proceedings Cryptographic Hardware and Embedded Systems, LNCS 2779, Springer-Verlag, pages 484 - 499, 2003.
- [IBM02] *IBM PCI cryptographic coprocessor*, May 2002. General Information Manual.
- [Iwam94] K. Iwamura, T. Matsumoto and H. Imai. *Montgomery modular multiplication method and systolic arrays suitable for modular exponentiation*. Electronic Communication. Japan, volume 77, No. 3, pages 40-50, 1994.

-
- [Izu03] T. Izu and T. Takagi. *Exceptional procedure attack on elliptic curve cryptosystems*. PKC 2003, LNCS 2567, Springer-Verlag, pages 224-239, 2003.
 - [Joye01] M. Joye and J. J. Quisquater. *Hessian elliptic curves and side-channel attack*. Proceedings Cryptographic Hardware and Embedded Systems, LNCS 2162, Springer-Verlag, pages 402-410, 2001.
 - [Kaih05] M. E. Kaihara and N. Takagi. *A hardware algorithm for modular multiplication/division*. IEEE Transactions on Computers, volume 54, No. 1, pages 12-21, January 2005.
 - [Kim04a] H. W. Kim, and S. Lee. *Design and implementation of a private and public key crypto processor and its application to a security system*, IEEE Transactions on Consumer Electronics. Volume 50, No. 1, pages 214-224, 2004.
 - [Kim04b] C. K. Kim, J. C. Ha, S. H. Kim, and S. Kim. *A secure and practical CRT-based RSA to resist side channel attacks*. ICCSA 2004, LNCS 3043, Springer-Verlag, pages 150-158, 2004.
 - [Kim05] C. K. Kim, J. C. Ha, S. J. Moon, S. M. Yen, W. C. Lien, and S. H. Kim. *An improved and efficient countermeasure against power analysis attacks*. IACR, Cryptography ePrint Archive, <http://eprint.iacr.org/2005/022.pdf>, 2005.
 - [Kits02] P. Kitsos, N. Sklavos, and O. Koufopavlou, *An efficient implementation of the digital signature algorithm*, 9th International Conference on Electronics, Circuits and Systems, volume 3, issue, pages 1151-1154, September 2002.
 - [Knut81] D. Knuth. *The Art of Computer Programming: Vol.2, Seminumerical Algorithms*. Addison-Wesley, 1981.
 - [Kobl87] N. Koblitz. *Elliptic curves in cryptosystem*. Math Comp. volume 48, pages 103-109, 1987.
 - [Koc98] C. K. Koc, and T. Acar, *Montgomery multiplication in $GF(2^k)$* , Designs, Codes and Cryptography, Kluwer Academic Publishers, Boston, volume 14, No. 1, pages 57-69, 1998.
 - [Koch85] M. Kochanski. *Developing an RSA chip*. Advances in Cryptology - CRYPTO'85, LNCS 218, Springer-Verlag, pages 350-357, 1985.
 - [Koch99] P. Kocher, J. Jaffe, and B. Jun. *Differential power analysis*. Advances in Cryptology. CRYPTO99: 19th Annual International Cryptology Conference, Santa Barbara, California, USA, pages 388-397, August 1999.
 - [Kore02] I. Koren, A. K. Peters, and Natick. *Computer Arithmetic Algorithms*. Prentice Hall, 2002.
 - [Korn94] P. Kornerup. *A systolic, linear-array multiplier for a class of right-shift algorithms*. IEEE Transactions on Computers, volume 43, issue 8, pages 892-898, August 1994.
 - [Lope99] J. Lopez and R. Dahab. *High performance single-chip FPGA Rijndael algorithm implementation*. Proceedings First Cryptographic Hardware and Embedded Systems, Springer-Verlag, LNCS 1717, pages 316-327, 1999.

-
- [Mami04] H. Mamiya, A. Miyaji, and H. Morimoto. *Efficient countermeasures against RPA, DPA and SPA*. Proceedings Cryptographic Hardware and Embedded Systems, LNCS 3156, Springer-Verlag, pages 343-356, 2004.
- [McLo04] M. McLoone, C. McIvor, and J. McCanny. *Montgomery modular multiplication architecture for public key cryptosystems*, IEEE Work shop on Signal Processing Systems, pages 349-354, October, 2004.
- [Mene93] A. J. Menezes. *Elliptic curve public key cryptosystems*. Kluwer Academic Publisher, 1993.
- [Mene97] A. Menezes, P. Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.
- [Mess99] T. S. Messerges and E. A. Dabbish and R. H. Sloan. *Power analysis attacks to modular exponentiation on smartcards*. Proceedings First Cryptographic Hardware and Embedded Systems, LNCS1717, Springer-Verlag, pages 144-157, 1999.
- [Mess02] T. S. Messerges and E. A. Dabbish and R. H. Sloan. *Examining smart-card security under the threat of power analysis attacks*. IEEE Transactions on Computers, volume 51, No. 5, pages 541-552, May 2002.
- [Mill85] V. Miller. *Use of elliptic curves in cryptography*. Proceedings Advances in Cryptology, Springer-Verlag, LNCS 0218, pages 417-426, 1985.
- [Moll01] B. Möller. *Securing elliptic curve point multiplication against side-channel attacks*. ISC 2001, LNCS 2200, Springer-Verlag, pages 324-334, 2001.
- [Mont85] P. Montgomery. *Modular multiplication without trial division*. Mathematics of Computation, volume 44, No.5, pages 19-21, 1985.
- [Mont87] P. Montgomery. *Speeding the pollard and elliptic curve methods of factorization*. Mathematics of Computation, volume 48, No. 177, pages 243-264, 1987.
- [Moon01] S. Moon, J. Park, and Y. Lee. *Fast VLSI arithmetic algorithms for high-security elliptic curve cryptographic applications*. IEEE Transactions on Computer Electronics, volume 47, No. 3, pages 700-708, 2001.
- [Nath07] P. Nathaniel and M. H. David. *Parallelized radix-4 scalable Montgomery multiplier*, BCCI'07, Rio de Janeiro, Brazil, pages 306-311, September 2007.
- [Nguy03] N. Nguyen, K. Gaj, D. Caliga, and T. El-Ghazawi. *Implementation of elliptic curve cryptosystems on a reconfigurable computer*. IEEE International Conference on Field-Programmable Technology (FPT) 2003, pages 60-67, 2003.
- [Nibo03] O. Nibouche, A. Bouridane, and M. Nibouche. *Architectures for Montgomery's multiplication*. IEEE Transactions on Computers, volume 52, No. 9, pages 361-368, 2003

- [Noza01] H. Nozaki, M. Motoyama, A. Shimbo, and S. Kawamura. *Implementation of RSA algorithm based on RNS Montgomery multiplication*. Proceedings Cryptographic Hardware and Embedded Systems, LNCS 2162 Springer-Verlag, pages 364-376, 2001.
- [Okad00] S. Okada N. Torii, K. Itoh and M. Takenaka. Implementation of Elliptic Curve Cryptographic Coprocessor over $GF(2^m)$ on an FPGA. Proceedings Cryptographic Hardware and Embedded Systems, LNCS 1965, pages 25-40, Springer-Verlag, 2000.
- [Okey00] K. Okeya and K. Sakurai. *Power analysis breaks elliptic curve cryptosystems even secure against the timing attack*. INDOCRYPT 2000, LNCS 1977, Springer-Verlag, pages 178-190, 2000.
- [Orla00] G. Orlando and C. Paar. *A high-performance reconfigurable Elliptic Curve processor for $GF(2^m)$* . Proceedings Cryptographic Hardware and Embedded Systems, LNCS 1965, pages 41-56, Springer-Verlag, 2000.
- [Orla01] G. Orlando and C. Paar. *A scalable $GF(p)$ Elliptic Curve processor architecture for programmable hardware*. Proceedings Cryptographic Hardware and Embedded Systems, LNCS 2162, pages 348-363, Springer-Verlag, 2001.
- [Orup95] H. Orup, *Simplifying quotient determination in high-radix modular multiplication*, Proceedings 12th Symposium on Computer Arithmetic, pages 193-199, 1995,
- [Piep03] J. Pieprzyk, T. Hardjono, and J. Seberry. *Fundamentals of Computer Security*. Springer-Verlag, 2003.
- [Posc95] K. C. Posch and R. Posch. *Modulo reduction in residue number systems*. IEEE Transactions on Parallel and Distributed Systems, volume 6, No. 5, pages 449-454, May 1995.
- [Rabi80] M. O. Rabin. *Probabilistic algorithm for testing primality*. Number Theory, volume 12, pages 128-138, 1980.
- [Rank87] G. Rankine. *THOMAS- a complete single chip RSA device*. Advances in Cryptology. CRYPTO'86, LNCS 263, pages 480-487, Springer-Verlag, 1987.
- [Rive78] R. L. Rivest, A. Shamir, and L. Adleman. *A method for obtaining digital signature and public-key cryptosystems*. Commun. ACM, volume 21, No. 2, pages 120-126, 1978.
- [Rive86] R. L. Rivest. *RSA chips (past/present/future)*. Advances in Cryptology - EUROCRYPT'84, LNCS 209, Springer-Verlag, pages 159-165, 1986.
- [Royo97] A. Royo, J. Moran, and J. C. Lopez. *Design and implementation of a coprocessor for cryptography applications*. European Design and Test Conference, pages 231-227, March 1997.

- [Sato98] T. Satoh and K. Araki, *Fermat quotients and the polynomial time discrete log algorithm for anomalous elliptic curves*, Commentarii Math. Univ. St. Pauli., volume 47, pages 81-92, 1998.
- [Sato03] A. Satoh and K. Takano. *A scalable dual-field elliptic curve cryptographic processor*. IEEE Transactions on Computers, volume 52, No. 4, pages 449-460, 2003.
- [Sava00] E. Savas, A. F. Tenca, and C. K. Koc. *A scalable and unified multiplier architecture for finite fields $GF(p)$ and $GF(2^n)$* . Proceedings Cryptographic Hardware and Embedded Systems, LNCS 1965, Springer-Verlag, pages 277-292, 2000.
- [SCIC04] *Security & Chip card ICs SLE88CFX4000P*, October 2004. short product information.
- [Stin02] D. Stinson, *Cryptography: Theory and Practice*. 2nd ed. Chapman and Hall / CRC, 2002.
- [Smar01] P. N. Smart. *The Hessian form of an elliptic curve*. Proceedings Cryptographic Hardware and Embedded Systems, LNCS 2162, Springer-Verlag, pages 118-125, 2001.
- [Su99] C. Y. Su, S. A. Hwang, P. S. Chen, and C. W. Wu. *An improved Montgomery's algorithm for high-speed RSA public-key cryptosystem*. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, volume 7, issue 2, pages 280-283, June 1999.
- [Suti98a] S. Sutikno, A. Surya, and R. Effendi. *Design and implementation of arithmetic processor over $F_{2^{155}}$ for elliptic curves cryptosystems*. IEEE Asia-Pacific Conference on Circuits and Systems, pages 647-650, 1998.
- [Suti98b] S. Sutikno, A. Surya, and R. Effendi. *An implementation of ElGamal elliptic curves cryptosystems*. IEEE Asia-Pacific Conference on Circuits and Systems, pages 483-486, 1998.
- [Syno] http://www.synopsys.com/products/logic/design_compiler.html
- [Tenc99] A. F. Tenca and C. K. Koc, *A scalable architecture for Montgomery multiplication*. Proceedings Cryptographic Hardware and Embedded Systems, LNCS 1717, Springer-Verlag, pages 94 -108, 1999.
- [Tenc01] A. F. Tenca, G. Todorov and C. K. Koc. *High-radix design of a scalable modular multiplier*. Proceedings Cryptographic Hardware and Embedded Systems, LNCS 2162, Springer-Verlag. Pages 185-201, 2001.
- [Tenc03] A. F. Tenca and C. K. Koc, *A scalable architecture for modular multiplication based on Montgomery's algorithm*, IEEE Transactions on Computers, volume 52, No. 9, pages 1215-1221, September 2003.
- [Tsoi03] K. H. Tsoi, K. H. Leung, and P. H. W. Leong. *Compact FPGA-based true and pseudo random number generators*. IEEE Symposiums on Field-Programmable Custom Machines, pages 1-11, 2003.

- [Vand90] A. Vandemeulebroecke, E. Vanzieleghem, T. Denayer and Paul G. A. Jespers. *A single chip 1024 bits RSA processor*. Advances in Cryptology - EUROCRYPT'89, LNCS 434, Springer-Verlag, pages 219-236, 1990
- [Walt93] C. D. Walter. *Systolic modular multiplication*. IEEE Transactions on Computers, volume 42, issue 3, pages 376-378, March 1993.
- [Wang05a] Y. Wang, J. Leiwo and T. Srikanthan, *A unified architecture for crypto-processing in embedded systems*, Second International Conference on Embedded Software and Systems, pages 37-43, December 2005.
- [Wang05b] Y. Wang, J. Leiwo and T. Srikanthan, *Efficient High Radix Modular Multiplication for High-speed Computing in Re-configurable Hardware*, IEEE International Symposium on Circuits and Systems (ISCAS), pages 1226-1229, May 2005.
- [Wolk02] J. Wolkerstorfer. *Dual-field arithmetic unit for $GF(p)$ and $GF(2^m)$* . Proceedings Cryptographic Hardware and Embedded Systems, LNCS 2523, Springer-Verlag, pages 500-514, 2002.
- [Yang98] C. C. Yang, T. S. Chang, and C. W. Jen. *A new RSA cryptosystem hardware design based on Montgomery's algorithm*. IEEE Transactions on Circuits and Systems: Analog Digital Signal Processing, volume 45, issue 7, pages 908-913, July 1998.
- [Zhen97] Y. Zheng. *Digital signcryption or how to achieve cost (signature & encryption) $\ll$ cost(signature) + cost(encryption)*. Advances in Cryptology - CRYPTO97, LNCS 1294, Springer-Verlag, pages 165-179, 1997.

Appendix

A. Elliptic Curve Arithmetic

In this section the five common forms of elliptic curve are presented.

A.1 General Form of EC

A curve defined over a field F is a collection of points $(x, y) \in F^2$, with F satisfying a simplified WeierstraB equation of the form [Piep03]:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \quad (\text{A.1})$$

Equation (A.1) defines a smooth elliptic curve such that the two partial derivatives of $f(x, y) = y^2 + a_1xy + a_3y - x^3 - a_2x^2 - a_4x - a_6$ do not vanish simultaneously, or $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial y} \neq 0$ at any point of the curve. The general form given by Equation (A.1) can be simplified.

$$y^2 = x^3 + ax + b \quad (\text{A.2})$$

where $4a^3 + 27b^2 \neq 0$. The formulas for addition and doubling operations are given below, where $P = (x_1, y_1)$, $Q = (x_2, y_2)$ and $P + Q = (x_3, y_3)$ are points on $E(F)$.

Addition Formula in simplified WeierstraB Form ($ECADD_{WE}(P, Q)$)

$$\begin{aligned} x_3 &= \lambda^2 - x_1 - x_2 \\ y_3 &= \lambda(x_1 - x_3) - y_1 \end{aligned}$$

where $\lambda = \frac{y_2 - y_1}{x_2 - x_1}$.

Doubling Formula in simplified WeierstraB ($ECDBL_{WE}(P)$)

$$\begin{aligned} x_3 &= \lambda^2 - 2x_1 \\ y_3 &= \lambda(x_1 - x_3) - y_1 \end{aligned}$$

where $\lambda = \frac{3x_1^2 + a}{2y_1}$.

A.2 Homogeneous Projective Coordinate

Projective coordinates are introduced for avoiding the division computation in the equation above [Chud86, Cohe98]. In the homogeneous coordinate, let $x = \frac{X}{Z}$ and $y = \frac{Y}{Z}$, then Equation (A.2) becomes:

$$Y^2Z = X^3 + aXZ^2 + bZ^3 \quad (\text{A.3})$$

If we let $P = (X_1, Y_1, Z_1)$, $Q = (X_2, Y_2, Z_2)$ and $P + Q = (X_3, Y_3, Z_3)$.

Addition Formula in Homogeneous Projective Coordinate ($ECADD_{ho}(P, Q)$)

$$X_3 = vA$$

$$Y_3 = \mu(v^2 X_1 Z_2 - A) - v^3 Y_1 Z_2$$

$$Z_3 = v^3 Z_1 Z_2$$

where $\mu = Y_2 Z_1 - Y_1 Z_2$, $v = X_2 Z_1 - X_1 Z_2$, $A = \mu^2 Z_1 Z_2 - v^3 - 2v^2 X_1 Z_2$.

Doubling Formula in Homogeneous Projective Coordinate ($ECDBL_{ho}(P)$)

$$X_3 = 2hs$$

$$Y_3 = w(4B - h) - 8Y_1^2 s^2$$

$$Z_3 = 8s^3$$

where $w = aZ_1^2 + 3X_1^2$, $s = Y_1 Z_1$, $B = X_1 Y_1 s$, $h = w^2 - 8B$.

A.3 Jacobian Projective Coordinate

In the Jacobian projective coordinate, let $x = \frac{X}{Z^2}$ and $y = \frac{Y}{Z^3}$, then Equation (A.3)

becomes:

$$Y^2 = X^3 + aXZ^4 + 6Z^6 \quad (\text{A.4})$$

If we let $P = (X_1, Y_1, Z_1)$, $Q = (X_2, Y_2, Z_2)$ and $P + Q = (X_3, Y_3, Z_3)$.

Addition Formula in Jacobian Projective Coordinate ($ECADD_{ja}(P, Q)$)

$$X_3 = -H^3 - 2U_1 H^2 + r^2$$

$$Y_3 = -S_1 H^3 + r(U_2 H^2 - X_3)$$

$$Z_3 = Z_1 Z_2 H$$

where $U_1 = X_1 Z_2^2$, $U_2 = X_2 Z_1^2$, $S_1 = Y_1 Z_2^3$, $S_2 = Y_2 Z_1^3$, $H = U_2 - U_1$, $r = S_2 - S_1$.

Doubling Formula in Jacobian Projective Coordinate ($ECDBL_{ja}(P)$)

$$X_3 = T$$

$$Y_3 = -8Y_1^4 + M(S - T)$$

$$Z_3 = 2Y_1 Z_1$$

where $S = 4X_1 Y_1^2$, $M = 3X_1^2 + aZ_1^4$, $T = 2S + M^2$.

A.4 Montgomery Form

Montgomery proposed the another form of elliptic curve in order to ease the addition requirements [Mont87]. Equation (A.5) shows the EC definition in Montgomery form.

$$by^2 = x^3 + ax^2 + x \quad (\text{A.5})$$

where $b(a^2 - 4) \neq 0$. In the projective coordinate of the Montgomery type method, the Y-coordinate is not used, so let $P = (X_1, Z_1)$, $Q = (X_2, Z_2)$, $P - Q = (X_3', Z_3')$ and $P + Q = (X_3, Z_3)$. The addition and doubling formulas of the Montgomery-type method are described as [Izu03, Fisc02]:

Addition Formula in Montgomery Form (ECADD_{mo}(P,Q))

$$X_3 = Z_3'(2(X_1Z_2 + X_2Z_1)(X_1X_2 + aZ_1Z_2) + 4bZ_1^2Z_2^2) - X_3'(X_1Z_2 - X_2Z_1)^2$$

$$Z_3 = Z_3'(X_1Z_2 - X_2Z_1)$$

Doubling Formula in Montgomery Form (ECDBL_{mo}(P))

$$X_3 = (X_1^2 - aZ_1^2)^2 - 8bX_1Z_1^3$$

$$Z_3 = 4(X_1Z_1(X_1^2 + aZ_1^2) + bZ_1^4)$$

A.5 Hessian Form

The Hessian elliptic curve is discussed in [Joye01, Smar01]. Equation (A.6) gives the elliptic curve in Hessian form on the $K = GF(q)$ field with $q \equiv 2 \pmod{3}$.

$$x^3 + y^3 + 1 = 3Dxy \quad (\text{A.6})$$

In projective coordinate, this becomes:

$$X^3 + Y^3 + Z^3 = 3DXYZ \quad (\text{A.7})$$

where $D \in K$ and $D^3 \neq 1$. Let $P = (X_1, Y_1, Z_1)$, $Q = (X_2, Y_2, Z_2)$ and $P+Q = (X_3, Y_3, Z_3)$.

Addition Formula in Hessian Form (ECADD_{he}(P,Q))

$$X_3 = Y_1^2 X_2 Z_2 - Y_2^2 X_1 Z_1$$

$$Y_3 = X_1^2 Y_2 Z_2 - X_2^2 Y_1 Z_1$$

$$Z_3 = Z_1^2 Y_2 X_2 - Z_2^2 Y_1 X_1$$

Doubling Formula in Hessian Form (ECDBL_{he}(P))

$$X_3 = Y_1 (Z_1^3 - X_1^3)$$

$$Y_3 = X_1 (Y_1^3 - Z_1^3)$$

$$Z_3 = Z_1 (X_1^3 - Y_1^3)$$

B. Signature, Encryption and Signcryption Schemes

$S=(S\text{Gen}, \text{Sig}, \text{Ver})$, $E=(E\text{Gen}, \text{Enc}, \text{Dec})$ and $SE=(SE\text{Gen}, \text{SigEnc}, \text{VerDec})$ represent the signature, encryption and signcryption schemes respectively, where the sender is s and the receiver is r . It is possible to unify these three schemes into a single general function F , as described in Algorithm B.1, as the procedures for the three schemes can be simplified to: key generation (KeyGen), including ($S\text{Gen}$, $E\text{Gen}$ and $SE\text{Gen}$), where $S\text{Gen}$ and $E\text{Gen}$ represent key generation for signing and encryption, respectively, and $SE\text{Gen}$ represents the key generation for both signing and encryption; signing encryption (SigEnc), including signing (Sig) and encryption (Enc); and verification decryption (VerDec) including verification (Ver) and decryption (Dec). Keys are generated as a private key set $\{SDK_s, VEK_s, SDK_r, VEK_r\}$ and a public key set $\{VEK_s, VEK_r\}$, where SDK represents the signing decryption key and VEK represents the verifying encryption key for the sender s and the receiver r , respectively. The function F , then has three different signcryption operations, S , E and SE , where m and c represent the plaintext and ciphertext, respectively.

Algorithm B.1 The general function of unifying signing, encryption and signcryption

-
1. $\text{KeyGen}\{SDK_s, VEK_s, SDK_r, VEK_r\}; \text{ pub} = \{VEK_s, VEK_r\}$
 2. Check Function (F)
 3. Switch (F) {
 - 3.1. S : $c = \text{Sig}_{\{SDK_s, \text{pub}\}}(m)$,
 $m = \text{Ver}_{\{SDK_r, \text{pub}\}}(c)$,
 - 3.2. E : $c = \text{Enc}_{\{SDK_s, \text{pub}\}}(m)$
 $m = \text{Dec}_{\{SDK_r, \text{pub}\}}(c)$,
 - 3.3. SE : $c = \text{SigEnc}_{\{SDK_s, VEK_s\}}(m)$,
 $m = \text{VerDec}_{\{SDK_r, VEK_r\}}(c).$
-